INTRODUCING ENHANCED FULLY-ADAPTIVE ROUTING DECISIONS WITHIN TORUS-MESH AND HYPERCUBE INTERCONNECT NETWORKS

by

CHRISTOPHER L LYDICK

B.S., Kansas State University, 2006

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Major Professor
Don Gruenbacher

Abstract

The method for communicating within an interconnection network, or fabric of connections between nodes, can be as diverse as are the applications which utilize them. Because of dynamic traffic loads on these interconnection networks, fully-adaptive routing algorithms have been shown to exploit locality while balancing loads and softening the effects of hotspots. One issue which has been overlooked is the impact of data traveling along the periphery of a selected minimal routable quadrant (MRQ) within these fully-adaptive algorithms. As data aligns with the destination in the x, y, and z dimensions for instance, the data then traverses the periphery of an MRQ. For each dimension that this occurs, the data is given one less choice for routing around hotspots which could appear later along the path. By weighting the decision of selecting a next-hop by avoiding the periphery of the selected MRQ, the data then has more options for avoiding hotspots. One hybridized routing algorithm which borrows heavily from CQR (an efficient and stable fully-adaptive algorithm), is introduced within this work. Enhanced CQR with Periphery Avoidance, attempts to weight the routing decision for a next hop using both output queues and the proximity to the periphery of the MRQ. This fully-adaptive algorithm is tested using simulations and a laboratory research cluster using a USB interconnect in the hypercube topology. It is also compared against other static, oblivious, and adaptive algorithms. Thor's Tack Hammer, the Kansas State University research cluster, is also benchmarked and discussed as an inexpensive and dependable parallel system.

Table of Contents

Table of Contents iv

List of Figures viii

List of Tables xii

Acknowledgements xiii

Dedication xiv

1 Introduction 1
- 1.1 Interconnection Networks . . . 1
- 1.2 High Performance Computing Clusters . . . 2
- 1.3 Routing within Interconnection Networks . . . 5
- 1.4 Contributions . . . 6

2 Previous Work 8
- 2.1 Undeliverable Data . . . 8
  - 2.1.1 Deadlock . . . 8
  - 2.1.2 Livelock . . . 9
  - 2.1.3 More on Undeliverable Data . . . 10
- 2.2 Static Routing Algorithms . . . 11
  - 2.2.1 Dimension Ordered Routing, DOR . . . 12
  - 2.2.2 Direction Ordered Routing, DIR . . . 12
- 2.3 Adaptive Routing Algorithms . . . 13
  - 2.3.1 Chaotic . . . 13
  - 2.3.2 Minimally Adaptive . . . 14
  - 2.3.3 Fully Adaptive: GOAL, GAL, CQR . . . 15
- 2.4 Oblivious Routing Algorithms . . . 17
  - 2.4.1 Valiant's Algorithm . . . 17
  - 2.4.2 Minimal Oblivious . . . 18
- 2.5 Traffic Patterns . . . 18
- 2.6 Latency and Throughput Analyses . . . 19

3 Enhancing Current Adaptive Routing Algorithms 21
- 3.1 Issues with Previous Algorithms . . . 21
- 3.2 Introducing Enhanced CQR with Periphery Avoidance . . . 22
  - 3.2.1 Path Diversity and Periphery Avoidance . . . 22

3.2.2 Periphery Avoidance Decision Function 24

4 The Research Cluster: Thor's Tack Hammer **28**
4.1 Topology 28
4.1.1 Subnetting and Addressing Scheme 30
4.2 Node and Cluster Specifications 35
4.3 Node and Cluster Performance 39
4.3.1 USB Interconnect Throughput 40
4.3.2 Linpack Benchmark 40
4.3.3 Power Consumption 42

5 Laboratory Results **44**
5.1 Methodology for Tests 44
5.1.1 A Packet's Progression through the Interconnection Network 46
5.2 Laboratory Results 48
5.2.1 Dimension Ordered Routing (DOR) Results 51
5.2.2 Direction Ordered Routing (DIR) Results 52
5.2.3 Minimal Oblivious Routing Results 52
5.2.4 Minimal Adaptive Routing Results 53
5.2.5 CQR Routing Results 53
5.2.6 Enhanced CQR Routing Results 54
5.2.7 Comparison of All Results 57

6 Simulation Results **59**
6.1 Methodology for Tests 59
6.2 Results from the 3-ary 3-cube Simulations 61
6.3 Results from the 4-ary 3-cube Simulations 64
6.4 Results from the 5-ary 3-cube Simulations 67
6.4.1 Comparison of All Results 70

7 Conclusions **71**
7.1 Future Work 73

References **76**

A Appendix A: Linux BASH Scripts **77**
A.1 USB Interconnect Scripts 77
A.1.1 file: interconnectUp 77
A.1.2 file: usbUp 79
A.1.3 file: usbPing 82
A.2 Data Analysis Scripts 84
A.2.1 file: interconnectTime 84
A.2.2 file: interconnectRouters 85
A.2.3 file: nodeSetup 86

A.2.4 file: interconnectThroughput . . . 87

B Appendix B: C Programs **91**
B.1 file: server.c . . . 91
B.2 file: loserver.c . . . 111
B.3 file: injector.c . . . 114
B.4 file: router.h . . . 116

C Appendix C: Matlab Simulation Scripts **117**
C.1 User Interface Scripts . . . 117
C.1.1 file: do_load.m . . . 117
C.1.2 file: view_load.m . . . 123
C.2 Routing Scripts . . . 125
C.2.1 file: dim2i.m . . . 125
C.2.2 file: i2dim.m . . . 126
C.2.3 file: dimord_route.m . . . 127
C.2.4 file: dirord_route.m . . . 129
C.2.5 file: minobliv_route.m . . . 132
C.2.6 file: do_minadaptive.m . . . 133
C.2.7 file: do_cqr1.m . . . 137
C.2.8 file: do_cqr2.m . . . 141
C.2.9 file: cqr_addPacket.m . . . 145
C.2.10 file: cqr_chooseQuadrant.m . . . 146
C.2.11 file: cqr_getRandomValue.m . . . 155
C.2.12 file: cqr_transformTrafficMatrix.m . . . 156

D Appendix D: Full Laboratory Results **157**
D.1 Matlab Plot Scripts . . . 157
D.1.1 file: convertLoad.m . . . 157
D.1.2 file: view_load.m . . . 159
D.1.3 file: i2dim.m . . . 159
D.1.4 file: dim2i.m . . . 159
D.2 Static Algorithms . . . 159
D.2.1 Dimension Ordered Routing Results . . . 159
D.2.2 Direction Ordered Routing Results . . . 163
D.3 Oblivious Algorithms . . . 167
D.3.1 Minimal Oblivious Routing Results . . . 167
D.4 Adaptive Algorithms . . . 171
D.4.1 Minimal Adaptive Routing Results . . . 171
D.4.2 CQR Routing Results . . . 175
D.4.3 Enhanced CQR Routing Results . . . 179

E Appendix E: Full Matlab Simulation Results 183
E.1 Results from the 3-ary 3-cube Topology . . . 183
E.1.1 Static Algorithms . . . 183
E.1.2 Oblivious Algorithms . . . 190
E.1.3 Adaptive Algorithms . . . 194
E.2 Results from the 4-ary 3-cube Topology . . . 206
E.2.1 Static Algorithms . . . 206
E.2.2 Oblivious Algorithms . . . 213
E.2.3 Adaptive Algorithms . . . 217
E.3 Results from the 5-ary 3-cube Topology . . . 229

List of Figures

1.1 A Torus . . . 3
1.2 4-ary 2-cube Slice of a Tori-Mesh Network . . . 4

2.1 Deadlock Example . . . 9
2.2 Livelock Example . . . 10
2.3 Minimal Adaptive Quadrants . . . 14
2.4 GAL Injection Queues . . . 16

3.1 The Ratio of Periphery Links to that of Non-Periphery Links . . . 25

4.1 Thor's Tack Hammer Node Connectivity . . . 29
4.2 An Addressing and Subnetting Example . . . 34
4.3 Thor's Tack Hammer Network Topology . . . 35
4.4 Single Node: The VIA VT310-DP . . . 36
4.5 Adaptec USB 2.0 Card used on Thor's Tack Hammer . . . 37
4.6 USB 2.0 NetLink Cables used on Thor's Tack Hammer . . . 38

5.1 An Example of Excellent Laboratory Results . . . 49
5.2 An Example of OK Laboratory Results . . . 50
5.3 An Example of Poor Laboratory Results . . . 51
5.4 CQR Routing (FL) TTH Results . . . 54
5.5 CQR Routing (FL-HS) - TTH Results . . . 55
5.6 Enhanced CQR Routing (FL) TTH Results . . . 56
5.7 Enhanced CQR Routing (FL-HS) TTH Results . . . 56

6.1 CQR Routing (FL) 3-ary 3-cube Simulation Results . . . 62
6.2 CQR Routing (FL-HS) 3-ary 3-cube Simulation Results . . . 62
6.3 Enhanced CQR Routing (FL) 3-ary 3-cube Simulation Results . . . 63
6.4 Enhanced CQR Routing (FL-HS) 3-ary 3-cube Simulation Results . . . 63
6.5 CQR Routing (FL) 4-ary 3-cube Simulation Results . . . 65
6.6 CQR Routing (FL-HS) 4-ary 3-cube Simulation Results . . . 65
6.7 ECQR Routing (FL) 4-ary 3-cube Simulation Results . . . 66
6.8 ECQR Routing (FL-HS) 4-ary 3-cube Simulation Results . . . 66
6.9 CQR Routing (FL) 5-ary 3-cube Simulation Results . . . 68
6.10 CQR Routing (FL-HS) 5-ary 3-cube Simulation Results . . . 68
6.11 ECQR Routing (FL) 5-ary 3-cube Simulation Results . . . 69
6.12 ECQR Routing (FL-HS) 5-ary 3-cube Simulation Results . . . 69

D.1 Dimension Ordered Routing (NN) TTH Results . . . 159

D.2 Dimension Ordered Routing (UR) TTH Results 160
D.3 Dimension Ordered Routing (BC) TTH Results 160
D.4 Dimension Ordered Routing (TP) TTH Results 161
D.5 Dimension Ordered Routing (TOR) TTH Results 161
D.6 Dimension Ordered Routing (FL) TTH Results 162
D.7 Direction Ordered Routing (NN) TTH Results 163
D.8 Direction Ordered Routing (UR) TTH Results 164
D.9 Direction Ordered Routing (BC) TTH Results 164
D.10 Direction Ordered Routing (TP) TTH Results 165
D.11 Direction Ordered Routing (TOR) TTH Results 165
D.12 Direction Ordered Routing (FL) TTH Results 166
D.13 Minimal Oblivious Routing (NN) TTH Results 167
D.14 Minimal Oblivious Routing (UR) TTH Results 168
D.15 Minimal Oblivious Routing (BC) TTH Results 168
D.16 Minimal Oblivious Routing - Transpose Traffic - TTH Results 169
D.17 Minimal Oblivious Routing (TOR) TTH Results 169
D.18 Minimal Oblivious Routing (FL) TTH Results 170
D.19 Minimal Adaptive Routing (NN) TTH Results 171
D.20 Minimal Adaptive Routing (UR) TTH Results 172
D.21 Minimal Adaptive Routing (BC) TTH Results 172
D.22 Minimal Adaptive Routing (TP) TTH Results 173
D.23 Minimal Adaptive Routing (TOR) TTH Results 173
D.24 Minimal Adaptive Routing (FL) TTH Results 174
D.25 Minimal Adaptive Routing - (FL-HS) - TTH Results 174
D.26 CQR Routing (NN) TTH Results 175
D.27 CQR Routing (UR) TTH Results 176
D.28 CQR Routing (BC) TTH Results 176
D.29 CQR Routing (TP) TTH Results 177
D.30 CQR Routing (TOR) TTH Results 177
D.31 CQR Routing (FL) TTH Results 178
D.32 CQR Routing (FL-HS) - TTH Results 178
D.33 Enhanced CQR Routing (NN) TTH Results 179
D.34 Enhanced CQR Routing (UR) TTH Results 180
D.35 Enhanced CQR Routing (BC) TTH Results 180
D.36 Enhanced CQR Routing (TP) TTH Results 181
D.37 Enhanced CQR Routing (TOR) TTH Results 181
D.38 Enhanced CQR Routing (FL) TTH Results 182
D.39 Enhanced CQR Routing (FL-HS) TTH Results 182

E.1 Dimension Ordered Routing (NN) 3-ary 3-cube Simulation Results 184
E.2 Dimension Ordered Routing (UR) 3-ary 3-cube Simulation Results 184
E.3 Dimension Ordered Routing (BC) 3-ary 3-cube Simulation Results 185
E.4 Dimension Ordered Routing (TP) 3-ary 3-cube Simulation Results 185

E.5 Dimension Ordered Routing (TOR) 3-ary 3-cube Simulation Results 186
E.6 Dimension Ordered Routing (FL) 3-ary 3-cube Simulation Results 186
E.7 Direction Ordered Routing (NN) 3-ary 3-cube Simulation Results 187
E.8 Direction Ordered Routing (UR) 3-ary 3-cube Simulation Results 187
E.9 Direction Ordered Routing (BC) 3-ary 3-cube Simulation Results 188
E.10 Direction Ordered Routing (TP) 3-ary 3-cube Simulation Results 188
E.11 Direction Ordered Routing (TOR) 3-ary 3-cube Simulation Results 189
E.12 Direction Ordered Routing (FL) 3-ary 3-cube Simulation Results 189
E.13 Minimal Oblivious Routing (NN) 3-ary 3-cube Simulation Results 190
E.14 Minimal Oblivious Routing (UR) 3-ary 3-cube Simulation Results 191
E.15 Minimal Oblivious Routing (BC) 3-ary 3-cube Simulation Results 191
E.16 Minimal Oblivious Routing (TP) 3-ary 3-cube Simulation Results 192
E.17 Minimal Oblivious Routing (TOR) 3-ary 3-cube Simulation Results 192
E.18 Minimal Oblivious Routing (FL) 3-ary 3-cube Simulation Results 193
E.19 Minimal Adaptive Routing (NN) 3-ary 3-cube Simulation Results 194
E.20 Minimal Adaptive Routing (UR) 3-ary 3-cube Simulation Results 195
E.21 Minimal Adaptive Routing (BC) 3-ary 3-cube Simulation Results 195
E.22 Minimal Adaptive Routing (TP) 3-ary 3-cube Simulation Results 196
E.23 Minimal Adaptive Routing (TOR) 3-ary 3-cube Simulation Results 196
E.24 Minimal Adaptive Routing (FL) 3-ary 3-cube Simulation Results 197
E.25 Minimal Adaptive Routing (FL-HS) 3-ary 3-cube Simulation Results 197
E.26 CQR Routing (NN) 3-ary 3-cube Simulation Results 198
E.27 CQR Routing (UR) 3-ary 3-cube Simulation Results 198
E.28 CQR Routing (BC) 3-ary 3-cube Simulation Results 199
E.29 CQR Routing (TP) 3-ary 3-cube Simulation Results 199
E.30 CQR Routing (TOR) 3-ary 3-cube Simulation Results 200
E.31 CQR Routing (FL) 3-ary 3-cube Simulation Results 200
E.32 CQR Routing (FL-HS) 3-ary 3-cube Simulation Results 201
E.33 Enhanced CQR Routing (NN) 3-ary 3-cube Simulation Results 202
E.34 Enhanced CQR Routing (UR) 3-ary 3-cube Simulation Results 202
E.35 Enhanced CQR Routing (BC) 3-ary 3-cube Simulation Results 203
E.36 Enhanced CQR Routing (TP) 3-ary 3-cube Simulation Results 203
E.37 Enhanced CQR Routing (TOR) 3-ary 3-cube Simulation Results 204
E.38 Enhanced CQR Routing (FL) 3-ary 3-cube Simulation Results 204
E.39 Enhanced CQR Routing (FL-HS) 3-ary 3-cube Simulation Results 205
E.40 Dimension Ordered Routing (NN) 4-ary 3-cube Simulation Results 206
E.41 Dimension Ordered Routing (UR) 4-ary 3-cube Simulation Results 207
E.42 Dimension Ordered Routing (BC) 4-ary 3-cube Simulation Results 207
E.43 Dimension Ordered Routing (TP) 4-ary 3-cube Simulation Results 208
E.44 Dimension Ordered Routing (TOR) 4-ary 3-cube Simulation Results 208
E.45 Dimension Ordered Routing (FL) 4-ary 3-cube Simulation Results 209
E.46 Direction Ordered Routing (NN) 4-ary 3-cube Simulation Results 210
E.47 Direction Ordered Routing (UR) 4-ary 3-cube Simulation Results 210

E.48 Direction Ordered Routing (BC) 4-ary 3-cube Simulation Results 211
E.49 Direction Ordered Routing (TP) 4-ary 3-cube Simulation Results 211
E.50 Direction Ordered Routing (TOR) 4-ary 3-cube Simulation Results 212
E.51 Direction Ordered Routing (FL) 4-ary 3-cube Simulation Results 212
E.52 Minimal Oblivious Routing (NN) 4-ary 3-cube Simulation Results 213
E.53 Minimal Oblivious Routing (UR) 4-ary 3-cube Simulation Results 214
E.54 Minimal Oblivious Routing (BC) 4-ary 3-cube Simulation Results 214
E.55 Minimal Oblivious Routing (TP) 4-ary 3-cube Simulation Results 215
E.56 Minimal Oblivious Routing (TOR) 4-ary 3-cube Simulation Results 215
E.57 Minimal Oblivious Routing (FL) 4-ary 3-cube Simulation Results 216
E.58 Minimal Adaptive Routing (NN) 4-ary 3-cube Simulation Results 217
E.59 Minimal Adaptive Routing (UR) 4-ary 3-cube Simulation Results 218
E.60 Minimal Adaptive Routing (BC) 4-ary 3-cube Simulation Results 218
E.61 Minimal Adaptive Routing (TP) 4-ary 3-cube Simulation Results 219
E.62 Minimal Adaptive Routing (TOR) 4-ary 3-cube Simulation Results 219
E.63 Minimal Adaptive Routing (FL) 4-ary 3-cube Simulation Results 220
E.64 Minimal Adaptive Routing (FL-HS) 4-ary 3-cube Simulation Results 220
E.65 CQR Routing (NN) 4-ary 3-cube Simulation Results 221
E.66 CQR Routing (UR) 4-ary 3-cube Simulation Results 221
E.67 CQR Routing (BC) 4-ary 3-cube Simulation Results 222
E.68 CQR Routing (TP) 4-ary 3-cube Simulation Results 222
E.69 CQR Routing (TOR) 4-ary 3-cube Simulation Results 223
E.70 CQR Routing (FL) 4-ary 3-cube Simulation Results 223
E.71 CQR Routing (FL-HS) 4-ary 3-cube Simulation Results 224
E.72 Enhanced CQR Routing (NN) 4-ary 3-cube Simulation Results 225
E.73 Enhanced CQR Routing (UR) 4-ary 3-cube Simulation Results 225
E.74 Enhanced CQR Routing (BC) 4-ary 3-cube Simulation Results 226
E.75 Enhanced CQR Routing (TP) 4-ary 3-cube Simulation Results 226
E.76 Enhanced CQR Routing (TOR) 4-ary 3-cube Simulation Results 227
E.77 Enhanced CQR Routing (FL) 4-ary 3-cube Simulation Results 227
E.78 Enhanced CQR Routing (FL-HS) 4-ary 3-cube Simulation Results 228
E.79 Minimal Adaptive Routing (FL) 5-ary 3-cube Simulation Results 229
E.80 Minimal Adaptive Routing (FL-HS) 5-ary 3-cube Simulation Results 230
E.81 CQR Routing (FL) 5-ary 3-cube Simulation Results 230
E.82 CQR Routing (FL-HS) 5-ary 3-cube Simulation Results 231
E.83 Enhanced CQR Routing (FL) 5-ary 3-cube Simulation Results 231
E.84 Enhanced CQR Routing (FL-HS) 5-ary 3-cube Simulation Results 232

List of Tables

2.1 Traffic Patterns Used to Test Routing Algorithms 18
2.2 Minimal Adaptive and Chaos Routing Algorithms Compared by Throughput 19
2.3 Minimal Adaptive and Chaos Routing Algorithms Compared by Latency . . 19
2.4 Adaptive Routing Algorithms Compared by Throughput and Latency 19

4.1 Linpack Test Results of Various Systems 41
4.2 Linpack/HPL Test Results of Thor's Tack Hammer 42
4.3 Power Consumption Results of Thor's Tack Hammer 42

5.1 Thor's Tack Hammer Results - Dimension Ordered Routing 52
5.2 Thor's Tack Hammer Results - Direction Ordered Routing 52
5.3 Thor's Tack Hammer Results - Minimal Oblivious 53
5.4 Thor's Tack Hammer Results - Minimal Adaptive 53
5.5 Thor's Tack Hammer Results - CQR Routing 54
5.6 Thor's Tack Hammer Results - Enhanced CQR 55
5.7 Comparison of All Laboratory Results 58

6.1 Simulation Results - 3-ary 3-cube - Average Utilizations 61
6.2 Simulation Results - 3-ary 3-cube - Standard Deviations 61
6.3 Simulation Results - 4-ary 3-cube - Average Utilizations 64
6.4 Simulation Results - 4-ary 3-cube - Standard Deviations 64
6.5 Simulation Results - 5-ary 3-cube - Average Utilizations 67
6.6 Simulation Results - 5-ary 3-cube - Standard Deviations 67

Acknowledgments

The success of this project could not have happened without the excellent supervision and support of Dr. Don Gruenbacher, who not only served as my major professor and advisor, but also encouraged me and inspired me to strive towards my goals. His selfless attitude and dedication towards academics is definitely appreciated.

I would also like to take this opportunity to thank my other advising professors, Dr. Caterina Scoglio and Dr. Sanjoy Das. Both of these individuals have given excellent guidance and support throughout my bachelor's and master's degrees.

Dedication

To my family.

Most notably, my sisters, my closest friends, and my Grammy. You all are the light in my life, my motivation, and the unending source of support that I've come to greatly appreciate. I love you all so very much.

Chapter 1

Introduction

Computational systems in the world today are being limited by a different factor than they once were. In the past, things such as gate switching speed, memory size, and thermal properties have been (and will always be) issues when aiming towards faster and more efficient computers. But, aside from those things, communication (both between systems and within them) has become a tremendous issue. The study of interconnection networks, or the fabric that connects various nodes within a system, aims at finding answers to these complex communication questions [8]. Then within these interconnection networks, one must examine the efficiency behind the actual data exchange, or routing. Once the routing has been covered, flow control must then be integrated to work well with the routing algorithms on a particular interconnection network [16].

The rest of the chapter will outline the specifics of interconnection networks, and will dive into a specific use of these networks within high performance computing clusters. Further more, the routing (which is a core discussion of this research work) is briefly introduced as are some of the initial results stemming from the evaluation of the routing algorithms.

1.1 Interconnection Networks

The realm of interconnection networks ranges tremendously, as do their applications. These interconnection networks can be viewed both macroscopically and microscopically. For instance, by zooming out and looking at the macro-level, the entire Internet can be viewed as an interconnection network with end-users and servers as the nodes, and the links, routers, and switches as the fabric connecting them. Different graphs or topologies of interconnection networks serve different purposes, and are very closely tied to their specific application. But within the different topologies there remains one quality which is central to all: balance the load as efficiently as possible, thus taking advantage of the diverse number of links, and increasing the overall ability for nodes to communicate.

At a more microscopic level, one can see that communication fabrics play a very large role in the efficiency of other parallel systems. For instance, the latest-and-greatest processors available through the popular manufacturers are called multi-core processors. These processors have multiple processing "cores" which attempt to compute or process data in parallel using a common communication fabric or interconnection network.

Though these two examples vary greatly in both the size of the communication fabric and the number of nodes within the network, both are solid examples of current applications of interconnection networks.

1.2 High Performance Computing Clusters

High Performance Computers (HPC), commonly referred to as "supercomputers" are generally used to process large simulations or tackle problems that were once considered impossible or improbable to solve in a non-parallel environment. HPCs are traditionally only found within companies or laboratories with large budgets because of their tremendous building and maintenance costs. The smaller clusters traditionally are thought of as computing clusters, as the HPC term tends to refer to the largest of these parallel systems. The largest (and fastest) 500 HPCs are tracked and cataloged within the Top500.Org website.

The applications on HPCs can and do vary widely. For example, corporations who manufacture automobiles could use an HPC or smaller cluster to effectively simulate prototype crash tests without having to physically build a prototype (which could only be good for one crash test). Instead, with a similar initial investment, these automotive corporations can crash these simulated prototypes as many times as they desire. They can also collect massive amounts of data from these simulations, helping the company's efficiency and vehicle safety.

Within HPCs, and more specifically, massively parallel processing computers (MPP), there exist many different ways to connect the nodes: butterfly networks, torus networks, non-blocking and blocking networks, among others. As intuition would state, the diverse number of interconnection possibilities matches the diverse applications for these HPCs. The work focused here centers on the network topology and interconnection fabric that is most popular within the HPC systems currently used by the US Department of Energy (US DOE) [4] [10]. This is a torus-mesh configuration.

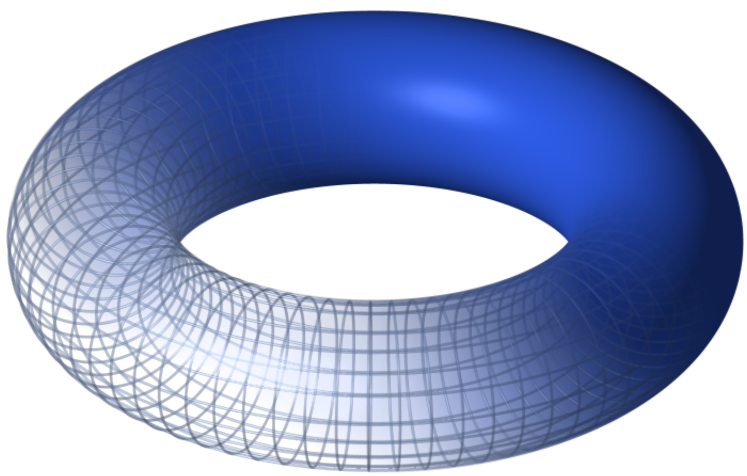


Figure 1.1: *Here is a torus shape, showing lines where the individual slices may reside. (Graphic public domain, http://en.wikipedia.org/wiki/Image:Torus.png.)*

The best way to explain this particular topology is to first consider a torus. A graphic of this shape is shown in Figure 1.1, but in layman's terms, it is shape often described as a product of two circles. Those familiar with tasty pastries like to compare it to a doughnut.

This network is composed of slices, connected in what one would consider in an X-Y plane. For the example in Figure 1.2, there are four nodes both in the X and Y directions.

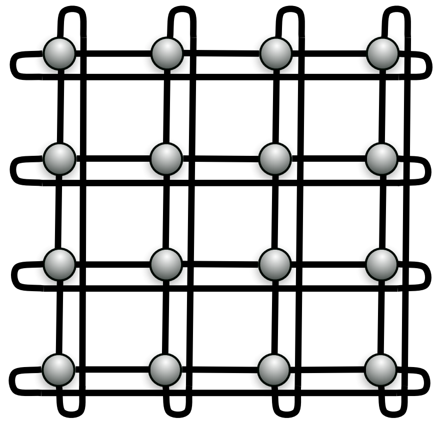


Figure 1.2: *This figure shows a 4-ary 2-cube slice of a tori-mesh network. These slices connect to other slices in an up/down (or +/- Z) direction to complete the torus shape.*

These nodes are also connected to other nodes in a Z direction. These slices continue to connect to other slices in the positive and negative Z directions and finally loop back around to the first, creating the torus shape. The defining characteristic that separates Tori-Mesh networks from that of hypercube networks is the irregular ratio between the number of slices in the Z direction and the number of nodes in the X and Y directions. If all three values are equal then that topology is a hypercube.

The Torus-Mesh interconnection network has many characteristics that make it attractive to the HPC community, and US DOE in general. One such characteristic is the high path diversity. Because of its configuration, there are generally many equivalent shortest paths between a single source and destination [8]. This improves a particular job or simulation's resiliency in being completed even when failures or hotspots occur. Resiliency is a quality which is of high importance within the US DOE on these large HPC systems [6].

Another quality that makes this particular network appealing to HPCs is the individual node's ability to quickly communicate with its neighbors with very little delay [8]. Most of the time when running simulations, nodes need only to communicate with their direct neighbors.

Lastly, and quite possibly most important, because of the tremendous investment in creating these systems, tori-mesh networks are generally easy to expand.

A very critical part of creating an efficient HPC is assigning efficient routing policies. This next section discusses just that.

1.3 Routing within Interconnection Networks

There are quite a few categories that routing algorithms in general fit into. Routing algorithms can be source or hop-based in that the decisions that data traverses can be determined at the source or intermittently through the network. Routing algorithms may also be dynamic or static in that the paths do or don't change throughout time. All routing algorithms within this research are hop-based and includes both static and dynamic implementations.

Routing within a Torus-Mesh network can be viewed within three separate categories. The first two are the dynamic algorithms, which causes the individual nodes (which traditionally also do the routing for other nodes' data in transit) to make decisions on-the-fly as to a packet's next hop. These decisions can be done *obliviously* (oblivious routing), or *adaptively* by examining various indicators of network availability (adaptive routing).

The last option for routing is referred to as a static approach. In this particular option, the data from source to destination travels a single path and usually cannot deviate from that path.

There are a couple of reasons why routing on this network (or any network for that matter) is so critical. While a network's topology is an important factor in the efficiency of communications, the routing mechanism is the limiting reagent in this mix: an efficient routing policy can make for a very efficient HPC, while a poor policy can make it very inefficient. Alongside with efficient communications, there is a need to evenly balance load between links and nodes within a network [1]. Again, a good routing policy will achieve this.

There are two large HPCs that will be frequently referenced within this work. The first, Sandia National Laboratory's Red Storm/Thor's Hammer MPP currently has 26,569 processing nodes, each with a 2.4 GHz dual-core processor and all nodes connected in a tori-mesh interconnect network. This HPC is ranked as the 6th highest performance computer in the world based upon the Linpack Benchmark (as of Nov 2007). This benchmark is a globally recognized standard for cataloging HPCs and is discussed later in this thesis.

Another well-known parallel computer within the HPC community is Lawrence Livermore National Laboratory's BlueGene/L MPP. This parallel system currently has 212,992 nodes, each with a PowerPC 440 700MHz processor. This system was ranked as the highest performing parallel computer in the world (as of Nov 2007).

Both rankings were taken from the independent group called Top500.org, which catalogs and tests the high-performance computers who wish to be ranked.

The Red Storm supercomputer at Sandia National Laboratories, as well as Blue Gene/L at Lawrence Livermore National Laboratories can implement any routing algorithm appropriate for their topologies, but research has pointed towards the stability of static routing, specifically an algorithm called Dimension Ordered Routing (DOR) [1] [2] [4]. Because of the complexity of these systems, along with the laboratories' experience with dynamic routing algorithms, stability tends to be one of the deciding factors when considering which algorithms to use.

Flow-control is one of the last pieces of the puzzle in terms of creating an overall efficient HPC. Unfortunately, the scope of this research work does not extend into that area.

1.4 Contributions

The work contained within this section briefly discusses and introduces the specific work of this research thesis and the contributions that accompany them.

Adaptive routing algorithms give a large amount of flexibility in two key areas of efficient HPC usage: proper load balancing, and the effective usage of a network's interconnect (thusly exploiting locality). These routing algorithms, which are introduced and discussed later in this thesis, have shown excellent characteristics for effective operation on hypercube and torus-mesh networks (among others).

An HPC system's overall reliability and availability is paramount in the efficient operation of large HPC systems [6]. Hotspots, which are naturally-occurring and unavoidable issues, occur when higher-than-average traffic demands are aimed at a single node. Multiple hotspots can occur within a network at any given time, and adaptive algorithms are the only algorithms which can adapt and respond to those traffic problems [5]. Therefore, this work aimed at increasing a routing algorithm's ability to make better decisions and to more effectively route around hotspots.

Along with introducing an enhanced decision function within these routing algorithms, a research cluster and simulation environment were created to verify those ideas. Each are introduced and explained in depth within this work.

An itemized list of contributions for this work include:

- Implement a simulation-based environment within Matlab to compare traffic and delay results for each algorithm given various hypercube topologies;
- Build a research cluster: a scaled version of the Red Storm/Thor's Hammer HPC;
- Implement a torus-mesh network on this research cluster using a USB interconnect;
- Implement various static, oblivious, and adaptive algorithms using a set of socket-layer C programs, validating and verifying previous work, and demonstrating this work's implementation differences;
- Introduce Enhanced CQR with Periphery Avoidance, which makes intermediate routing decisions by routing productively but avoiding the perimeter when possible;
- Discuss the impact of this new enhancement and demonstrate evidence of its potential.

Chapter 2

Previous Work

This chapter will attempt to provide a well-rounded literature review of the focused area of this thesis. Next is a discussion on undeliverable data – a critical issue with routing algorithms. Finally there is a discussion of both static and dynamic routing algorithms as well as evidence of their reported performance.

2.1 Undeliverable Data

With the high path diversity and the large number of routing decisions within tori-mesh networks, these issues of livelock and deadlock are of great concern and must be considered within each specific routing algorithm. These next two sections briefly introduce the problem of undeliverable data.

2.1.1 Deadlock

Deadlock, as it is defined by [8], occurs when a set of agents holding resources are waiting on another set of resources such that a cycle of waiting agents is formed. Most networks are designed to avoid deadlock, but it is also possible to recover from deadlock by detecting and breaking cyclic wait-for relationships.

Deadlock is more of an issue on routing algorithms that do not allow for misrouting of data. When this data reaches a node or link it cannot traverse through, it stops its progress, and is usually dropped [9]. Because of finite buffers, and random failures, deadlock is usually an issue to be considered within routing algorithms. This particular issue can also be subdued by implementing a few policies. As outlined within [9], deadlock prevention, avoidance, and recovery techniques can all reduce the problem of deadlock.

Figure 2.1 shows a deadlock example.

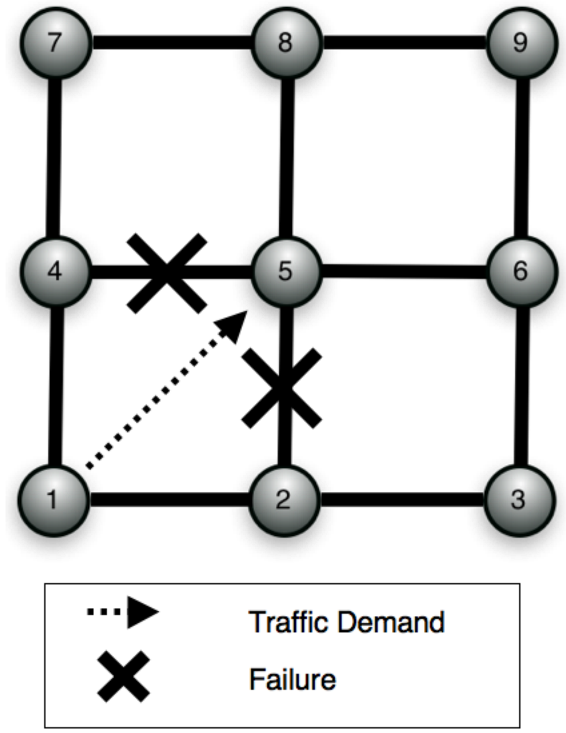


Figure 2.1: *This example shows how deadlock occurs, using the same traffic demand as before but this data is unable to reach the destination because of its inability to misroute. This data will end up being dropped at node 4 or node 2 after being unable to reach the destination.*

2.1.2 Livelock

Livelock, as it is defined within [8], occurs when a packet is not able to make progress in the network and is never delivered to its destination. Unlike Deadlock, though, a livelock packet continues to move through the network.

Livelock tends to occur most frequently when data is allowed to be misrouted, or routed non-minimally [9]. More specifically, livelock becomes a reality when data is forever in transit towards its destination, and continues to move within the network, but does not reach the destination within finite time. Usually this issue comes about because of over-utilized channels or link failures. There are two schemes towards tackling the issue of livelock within routing algorithms. By using only minimal paths, and having great restrictions on non-minimal paths, livelock can be less of an issue [9].

Figure 2.2 shows a livelock example.

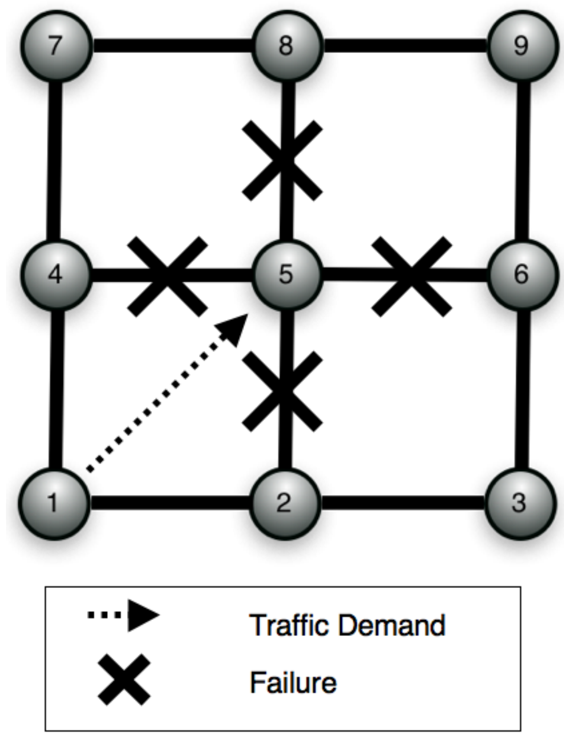


Figure 2.2: *A simple livelock example showing a traffic demand from node 1 to node 5. The traffic is unable to reach the destination, regardless of its ability to misroute up to node 8, or node 6. This data will end up being misrouted indefinitely around the destination.*

2.1.3 More on Undeliverable Data

Deadlock and livelock are two issues playing tug-of-war with one another. By reducing an algorithm's likelihood of falling into deadlock, it increases its likelihood of being vulnerable to livelock, and vice versa. These inversely-related issues are of great importance and will be part of the discussion of each algorithm listed within this chapter.

2.2 Static Routing Algorithms

As it was stated before there is some appeal to static routing algorithms within these large parallel systems. Because of their simplicity, these algorithms tend to be very easy to implement within hardware [8]. As the name would imply, the path that data takes from a source to destination will not change through time. The difference between the algorithms depends on how these static routes are chosen. But once chosen, the path from source to destination cannot change.

While static, or source-based routing may sound simple, it is really only efficient when the administrator or operator knows a general idea of traffic patterns prior to any operations. Under those ideal conditions, source-based routing can very effectively balance traffic throughout the network. In terms of simulations, it would not be too terribly difficult to determine these traffic patterns prior to starting the simulations - thus, these routing techniques seem to be a pretty good choice on these HPCs.

As always, there are never any free lunches. Assuming that the operator has correctly predicted a traffic pattern and has generated all routing tables for a given set of nodes which will be used for a simulation, all notion of stability goes out the window when one of two things occur: (1) a node loses its ability to communicate with a set of nodes, or (2) a link becomes over- utilized, rendering it unusable. These two topics form the foundation of this research which are referred to as nodal failures and hot-spots, respectively.

Both of the following static approaches were initially developed because of their ease and simplicity in terms of hardware implementation [9]. It's no wonder why they were initially attractive to large HPC systems.

2.2.1 Dimension Ordered Routing, DOR

The most published application of static or source-based routing algorithms within these DOE HPC systems is what is called Dimension Ordered Routing (DOR) [1] [8] [9]. At the source, the node pre-computes a set of preferred directions to route the packet within. For instance, if a source and destination pair exist within the same Z dimension, only differing within the X and Y dimensions, it would be preferred not to change coordinates within the Z dimension [8] [9].

Once the preferred directions have been calculated, this information is placed within the header of the data and it is directed into the corresponding X direction first. Once the packet has successfully reached the same X coordinate as the destination, it then routes in the Y direction (if it needs to). Following the Y direction, intermediate nodes then continue to route in the Z direction (again, if necessary) until it finally reaches the destination.

So, in the case above, if the pre-computed preferred directions were $[+X, -Y, 0]$, this packet would fully route in the positive X direction and then the negative Y direction last. In terms of all possible directions, this is the order that data is routed within the network: $[+X, -X, +Y, -Y, +Z, -Z]$.

Dimension Ordered Routing is proven to be free from deadlocks [7] by balancing the load on virtual channels. The issue shown in [7] demonstrates that by making this modification the structure of the network and the diversity of usable channels changes. This has shown to create non-uniform patterns. These virtual channels are only necessary on torus or wrap-around networks, so with regular mesh-type networks, the full benefits of DOR can be witnessed.

2.2.2 Direction Ordered Routing, DIR

Direction Ordered Routing (DIR) is very similar to that of Dimension Ordered Routing. The difference in these two resides in the order in which the routing occurs once the preferred directions have been calculated. These calculations are done just as they are with DOR, but instead of fully routing within the X direction before progressing to the Y direction and so on, DIR routes positively within all directions first, then routes negatively within the directions last [8] [9]. The order that data is routed within the network is as follows: $[+X, +Y, +Z, -X, -Y, -Z]$

The same issues with regard to deadlock apply to that of DIR as they do with DOR.

2.3 Adaptive Routing Algorithms

Adaptive routing algorithms seem to be where the most recent research has been heading in terms of these HPC systems. Though there is some stigma with regard to their overall stability within complex networks, it is the work of this research to try to show highlights of using these routing algorithms. The balance between intuition of network conditions and making routing decisions based on that intuition is definitely taken at the expense of simplicity.

2.3.1 Chaotic

Chaotic routing uses what is called deflection routing [8], randomly granting contending data access to channels and deflecting (or misrouting) the data that is denied access to the requested channel. These particular deflections may or may not be productive (meaning an extra hop may be incurred by deflection).

Deadlock is not as much of an issue for this algorithm, as the data can be easily misrouted [5] [8]. Livelock can be handled within this routing scheme by introducing time stamps and battle scars [5]. Time stamps indicate when the packets were introduced into the network, and when contending packets request the same channel, the router can grant the packet with the oldest packet. Battle scar implementations require packets to contain information as to how many times they have been misrouted. Again, the router can grant packets with larger battle scars access to channels over those with smaller or zero battle scars [5].

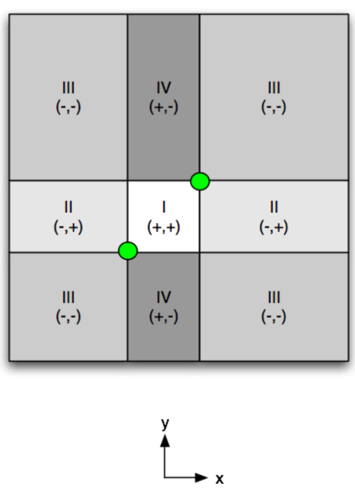
2.3.2 Minimally Adaptive

The minimal adaptive routing algorithm first starts the data progression by designating the minimal quadrant, which contains all possible shortest paths from source to destination. This quadrant is of size $\prod^{k} \Delta k$, where k is the number of dimensions which source and destination differ by greater than zero. Any node within this quadrant is considered to be a possible intermediate node along some of many possible shortest paths.

Figure 2.3.2 shows the an example on a $n = 2$ torus network. Quadrant I is the most minimal because it contains all minimal paths. Once this minimal quadrant has been properly identified, all routing occurs progressively within it.

For instance, if $s = (2, 3)$ and $d = (5, 6)$, a packet must route minimally within Quadrant I by either routing positively in the X direction or positively in the Y direction. The decision as to whether to pick the X or Y direction depends on local characteristics such as output queue length. Once a next hop has been determined, the data is then routed to the next hop, and the process repeats until it reaches the destination.

Figure 2.3: *This figure shows the possible quadrants and the distances data must traverse while within them. Quadrant I is the most minimal, if Δx and Δy are equal, then quadrants II and IV are the same. Quadrant III is the least minimal in this example. (Figure adapted from [12].)*



Once this quadrant has been determined, routing is done within it by choosing the next progressive hop by analyzing some local measurements or congestion indicators - typically output queue length [8].

Typically this particular algorithm fares well towards local load balancing, but has no insight of congestion further down the path, limiting its ability to achieve global load balancing. By using virtual-channels, deadlock can be avoided [3].

2.3.3 Fully Adaptive: GOAL, GAL, CQR

The essence of the best algorithms is epitomized within the characteristics of fully adaptive routing algorithms for these networks [13]. By exploiting locality and the network's wrap-around feature, an efficient routing algorithm should be able to have the flexibility to misroute data to balance the traffic load. Fully adaptive algorithms do just this.

The work that was done by Singh, et. al, [12] [13] [14], has collectively progressed into what seems like a solid attempt at finding a stable, reliable, and fully adaptive routing algorithm on tori-mesh networks for a diverse set of traffic patterns.

The first of the three algorithms, Globally Oblivious Adaptive Locally (GOAL), works very similarly to that of Minimal Adaptive with a few changes. First, it finds all possible routable quadrants and scales the probability of selecting a particular quadrant based on how minimum it is with respect to a particular source/destination pair. Once those probabilities are assigned and a particular quadrant has been selected, routing is done minimally within it [12]. This algorithm uses a virtual-channels implementation (called *-channels) to ensure deadlock and cycle-free operation [12].

One problem with this algorithm, which was a similar problem with Minimal Adaptive, was its inability to recognize global congestion. The answer to this comes through in the next algorithm that was developed by this same group, Globally Adaptive Load-Balanced (GAL).

This algorithm is implemented by keeping a globally visible set of input queues for each node, called injection queues. Each node has as many injection queues as it has inputs, and when a packet begins its commute from source to destination, the injection queue on the destination node relative to the link the packet will be received on is increased. As with GOAL, GAL first determines the routable quadrant, then selects this quadrant based on distance (as did GOAL), but it also considers the injection queues. This way, nodes can keep a general idea of how much total traffic is being sent in any particular direction, and can attempt to balance that out globally [14].

Figure 2.4: *A graphical representation of how GAL Injection queues appear globally. The value increments as packets are sent toward a particular interface, and they decrease as packets reach a destination through a particular interface.*

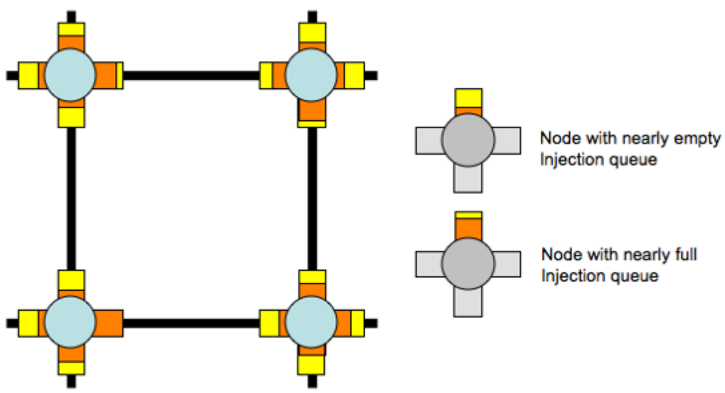


Figure 2.4 shows graphically how these injection queues may look.

Another versatile option for this algorithm to reduce its complexity is by subnetting the injection queues. By grouping multiple interfaces together with a single queue, thus reducing the number of queues to store globally, the authors showed that this technique had little impact on the overall performance of this algorithm [14]. This algorithm is also said to be deadlock and cycle-free [14].

Finally, a better meld of the global and local qualities that GAL and GOAL provided was integrated into a simple-to-implement and efficient protocol called Adaptive Channel Queue

Routing (CQR) [13]. This routing algorithm uses local information in the form of output queues (as opposed to the GAL algorithm which implemented input injection queues which were globally accessible). CQR uses these output queues as estimators for global congestion, which was proven within the work of [13]. The work showed that even in the absence of global information, local information can provide good estimations of global congestion.

Also, relieving the algorithm from accessing and changing global information made it much simpler to implement (not to mention lower its overhead). It was also shown to quickly adjust to changing traffic patterns and the changeover from minimal routing to non-minimal routing[13]. CQR was also shown to be deadlock and cycle-free [13].

2.4 Oblivious Routing Algorithms

This final section of routing algorithms within these HPC networks is referred to as Oblivious Routing. These routing algorithms tend to hybridize aspects of the adaptive and static algorithms. For instance, oblivious routing techniques tend to keep hardware implementations simpler than adaptive algorithms (a static characteristic), while the paths that data takes within the network can change throughout time (which is an adaptive characteristic).

The next subsections introduce two popular oblivious algorithms, Valliant's Algorithm and Minimal Oblivious.

2.4.1 Valiant's Algorithm

Valiant's Algorithm came about in an attempt to balance loads within any particular topology [8]. The algorithm works by first selecting at random an intermediate node, x, and once successfully routing to that intermediate node, then routing from x to the destination. As it's stated, any arbitrary routing algorithm can be used to get data from source to the intermediate node and then from the intermediate node to the destination [8].

But with the addition of better network load balance, locality is clearly taken out of the picture [12]. And as it was stated before, the DOE HPCs have a desire to maintain locality

Name	Description
Nearest Neighbor (NN)	Nodes send data to only their neighbors with equal probability.
Uniform Random (UR)	Nodes send data to random destinations.
Bit Compliment (BC)	(x, y, z) sends data to $(k - x - 1, k - y - 1, k - z - 1)$.
Transpose (TP)	(x, y, z) sends to all possible permutations of x,y, and z.
Tornado (TOR)	(x, y, z) sends to $(x + \frac{k}{2} - 1, y, z)$.
Flood (FL)	(x, y, z) sends to all other nodes within the network.
HotSpot (HS)	n number of nodes have k times more traffic demand than others.

Table 2.1: *These are the traffic patterns outlined in [12] [13] [14], which have been used to validate and verify routing algorithms on hypercube and torus mesh networks.*

because of the nature of the applications. This makes Valiant's Algorithm a possibly good application in other networks, but not within Redstorm, BlueGene/L, or the like.

Deadlock is said to be avoided using two subnetworks for the two steps, and using two virtual channels within each step [15].

2.4.2 Minimal Oblivious

The Minimal Oblivious algorithm works very closely to that of the Minimal Adaptive algorithm listed above, in that the traffic will traverse one of many possible minimum shortest paths. The algorithm first calculates its minimal quadrant, and then selects a random intermediate node somewhere within the minimum quadrant. Once that intermediate node is selected, the data is routed first to the intermediate node and secondly from the intermediate node to the destination, all done randomly, minimally, and oblivious to any network indicators or local conditions [8].

2.5 Traffic Patterns

In order to analyze these above algorithms, there are a few traffic patterns that are used to test and validate an algorithm's ability to route packets across a given topology effectively. These traffic patterns usually serve as a worst-case scenario for traffic demands, but they are real-world possibilities for demands none-the-less.

Traffic Pattern	Minimal Oblivious	Chaos Routing
UR	0.6	1.0
BC	0.3	0.5
TP	0.5	0.5
HS	0.6	0.9

Table 2.2: *This table shows results obtained from [5], comparing the Chaos and Minimal Adaptive routing algorithms with a 100% load applied the simulated hypercube network. The results show the fraction of data successfully transmitted given the traffic pattern at 100% load.*

Traffic Pattern	Minimal Oblivious	Chaos Routing
UR	600 (cycles)	550 (cycles)
BC	2200	1000
TP	690	900
HS	600	500

Table 2.3: *This table shows the results obtained from the same conditions as in Table 2.2. This table shows the latency (in terms of extra cycles) seen when observing the throughput in the previous table.*

2.6 Latency and Throughput Analyses

This section attempts to show previous work, comparing the algorithms of interest in terms of throughput and latency. All algorithms were compared using similar approaches, by analyzing how well the algorithms reacted given specified traffic demands.

Tables 2.2 and 2.3 show useful data taken from [5], comparing the latency and throughput

Traffic Pattern	Minimal Adaptive	GOAL	GAL	CQR
NN/UR	1.0	0.75	1.0	1.0
TR/BC/TOR	0.33	0.53	0.53	0.53
Average	0.63	0.67	0.73	0.7
HS	0.46	0.48	0.49	0.49
Low Load Latency	4.45	6.17	4.45	4.45

Table 2.4: *This table compares the adaptive routing algorithms in terms of throughput (shown in fractions given 100% loads), as well as latency during low loads. This information was obtained from [13] [14] [12].*

of Minimal Oblivious and Chaos routing algorithms. The results show Chaotic routing meeting or exceeding the performance of Minimal Oblivious.

Table 2.4 compares the adaptive algorithms in terms of throughput. All three of the Stanford group's algorithms show consistent progression over that of Minimal Adaptive. CQR ends up being the most resilient algorithm in terms of these chosen traffic demands when concerning throughput and latency.

Chapter 3

Enhancing Current Adaptive Routing Algorithms

Because adaptive algorithms are able to exploit locality as well as misrouting data to balance the load, they are very attractive algorithms for implementation. Their stability is necessary for traffic changes and CQR, Minimal Adaptive, and other fully adaptive algorithms alike are able to transition between those traffic differences well [13].

3.1 Issues with Previous Algorithms

As it was stated before, adaptive algorithms have two qualities which make them attractive. They have the ability to take advantage of locality while capitalizing on path diversity by balancing load. As for CQR, once a quadrant has been selected at the source, the data in transit is not allowed to deviate outside of that quadrant and must make productive moves towards the destination [13].

A productive move is defined as intermediately choosing a next hop so that the distance between the data in transit and the intended destination decreases [13]. If the distance stays the same or increases, it cannot be considered a productive hop.

Adaptive algorithms such as CQR could more effectively route around hotspots or nodal failures if they encountered these network abnormalities away from the edges of the selected quadrants. In fact for every "edge" of the quadrant the data hits, it reduces the possible decisions it can make further down the path.

Take for instance a simple example where a selected quadrant has $\Delta x = 4$ and $\Delta y = 2$. If the first two hops within that quadrant are in the y-direction, then the final four hops must be in the x-direction. If the data encounters a hotspot along the x-direction, it has no choice but to continue routing through the congestion. The next section introduces the concept further, called Periphery Avoidance.

3.2 Introducing Enhanced CQR with Periphery Avoidance

Along with the flexibility that adaptive routing algorithms offer, there are some drawbacks. Minimal Adaptive, for instance, offers great flexibility in routing decisions near the center of the selected quadrant, while offering less flexibility near the periphery of the quadrants. This is because once data has reached the periphery of its intended routing quadrant (it can also be thought of as aligning in a similar dimension with the destination), it reduces the ability to route around congestion and hotspots. By offering the traffic a path away from the periphery of the selected quadrant, we increase data's ability to adaptively route around hot-spots.

Analytically speaking, a majority of the paths within a quadrant fall along the periphery of the network, but many choices do exist within the interior of a quadrant. The next section discusses path diversity as it relates to this overall concept of periphery avoidance and introduces a decision function in which queue sizes are weighted with the distance from the periphery of a quadrant to choose a next-hop.

3.2.1 Path Diversity and Periphery Avoidance

The work from [8] outlines mathematical expressions for calculating the number of possible distinct routes within a torus-mesh or hypercube network. Below is the expression from [8] for a 2-Dimensional network:

$$|R_{sd}| = \binom{\Delta x + \Delta y}{\Delta x} \tag{3.1}$$

And for a 3-Dimensional network, the expression for finding the number of possible distinct routes between source and destination is:

$$|R_{sd}| = \binom{\Delta x + \Delta y + \Delta z}{\Delta x} \cdot \binom{\Delta y + \Delta z}{\Delta y} \tag{3.2}$$

Given the possible routes that exist between source and destination within a quadrant for the values for Δx, Δy, and Δz, a subset of these routes use the interior of the network while another disjoint subset use one or more parts of the periphery. The next equation was created to demonstrate that relationship:

$$|R_{sd}| = |R_{sd\bar{p}}| + |R_{sdp}| \tag{3.3}$$

The expression for $|R_{sdp}|$ describes the routes which use the periphery during one or more hops, while $|R_{sd\bar{p}}|$ describes the routes which route only within the interior of the quadrant.

It was easier to think in terms of counting the paths which use the interior of the quadrant than developing an expression for counting the number of paths which use the periphery of the quadrant. For a 2-Dimensional network, that expression is:

$$|R_{sd\bar{p}}| = \binom{(\Delta x - 1) + (\Delta y - 1)}{(\Delta x - 1)} + 2 \cdot n \tag{3.4}$$

And for a 3-Dimensional network the expression is:

$$|R_{sd\bar{p}}| = \binom{(\Delta x - 1) + (\Delta y - 1) + (\Delta z - 1)}{(\Delta x - 1)} \cdot \binom{(\Delta y - 1) + (\Delta z - 1)}{(\Delta y - 1)} + 2 \cdot n \tag{3.5}$$

An example 2-D network with a quadrant of size $\Delta x = 4$ and $\Delta y = 3$ demonstrates the relationship between these values. Therefore, given those values and using Equations 3.3 and 3.4, the counts for the various routes are:

$$|R_{sd}| = \binom{4+3}{4} = 35$$
$$|R_{sd\bar{p}}| = \binom{3+2}{3} + 2 \cdot 2 = 14$$
$$\therefore$$
$$|R_{sdp}| = 21$$

This example depicts the general trend of the relationship between $|R_{sd}|$, $|R_{sd\bar{p}}|$, and $|R_{sdp}|$. Because of the high number of links which utilize areas of the periphery, and given that hot-spots are common occurrences avoiding these links whenever possible should definitely improve the data's ability to route around localized hotspots when encountering them.

Figure 3.1 shows the relationship between the number of links which utilize the periphery and the number of links that do not.

3.2.2 Periphery Avoidance Decision Function

Traditional CQR uses two decision phases when sending data through the network. The first phase is done at the source, when the quadrant to route within is selected. Once this decision has been made, the data carries with it a vector of size n (e.g. $v = \{x_1, x_2, ...x_n\}$), referred to as the minimal direction vector. The values within this vector indicate whether routing is allowed within a given dimension. If a particular dimension carries a value of $+1$, then the data may route in the positive direction in that dimension. Similarly, if the value is -1, the data may route in the negative direction of that dimension. Finally, if the value is 0, routing is not allowed in that dimension.

For this discussion, the minimal direction vector must not only contain the allowed directions to route within that dimension, but must also contain the Δ values for that dimension. This enables the decision function, which is explained below, to weight the links near the center of the quadrant higher than those which reside on the periphery. As the

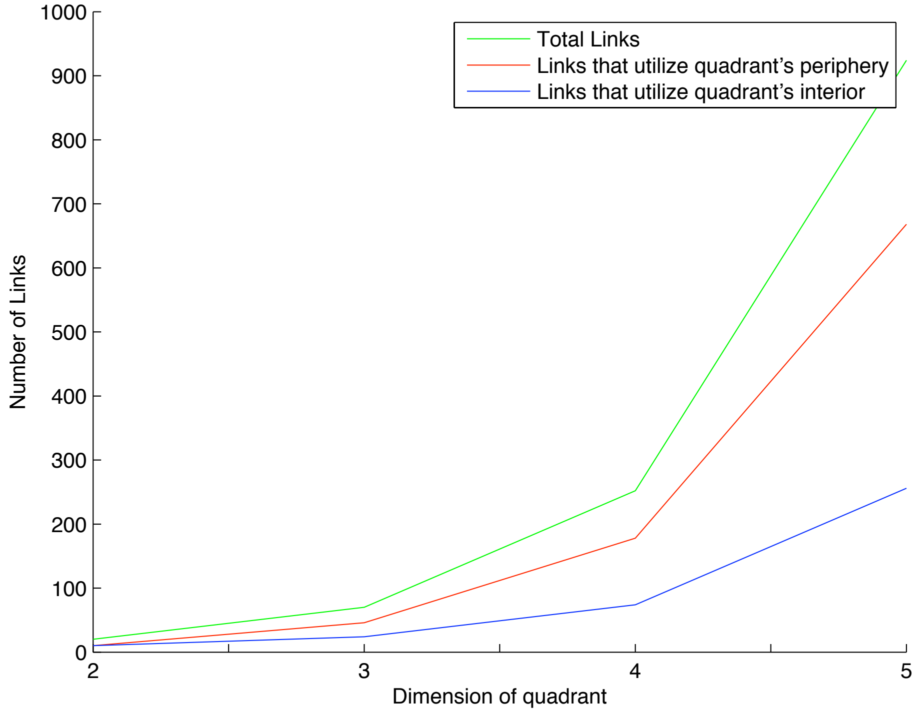


Figure 3.1: *This figure demonstrates the ratio between the links which utilize the periphery of a selected quadrant to that of the links which utilize the interior of the quadrant. As the quadrant size increases, the ratio converges to 4:1.*

data progresses through the network and decides a next hop, this vector must be modified to reflect the actual Δ's at each intermediate hop. This enables the decision function to constantly evaluate and aim towards keeping the data in transit away from the periphery of the network.

The Enhanced CQR decision function as it relates to the concept of Periphery Avoidance uses the same first phase as traditional CQR, but uses a slightly different method for choosing intermediate nodes within a network on the second phase. Before considering this enhanced version, first consider the old decision function for CQR:

$$Next\_Hop = min(Q(x_1), Q(x_2), ...Q(x_n)) \forall x_i \in n \tag{3.6}$$

Where $Q(x_i)$ is:

$$Q(x_i) := k \text{ s.t. } \begin{cases} k = \text{Output queue value at direction } x_i \text{ for } \Delta x_i > 0 \\ k = +\infty \text{ for } \Delta x_i = 0 \end{cases} \tag{3.7}$$

The function $Q(x_i)$ returns $+\infty$ when x_i is not a productive direction, otherwise, it returns the value of the output queue in the dimension x_i. Therefore, a selection is made to ensure the next hop is productive, and relative to the shortest output queues.

To create a decision function which incorporates both output queue length and periphery avoidance, the following equation was developed:

$$Next\_Hop = min(Q_p(x_1), Q_p(x_2), ...Q_p(x_n))\forall x_i \in n \tag{3.8}$$

Where $Q_p(x_i)$ is:

$$Q_p(x_i) := k \text{ s.t. } \begin{cases} k = (Q(x_i) + 1) \cdot (1 - \frac{\Delta i}{\Delta total}) \text{ for } \Delta x_i > 0 \\ k = +\infty \text{ for } \Delta x_i = 0 \end{cases} \tag{3.9}$$

The value for $\Delta total$ is the sum of all the absolute values in each dimension. One is added to the queue lengths to ensure that zero-length queues do not compromise the function's ability to avoid the periphery.

First, consider a selected quadrant with direction vector $v = \{\Delta x = 4, \Delta y = 10, \Delta z = 0\}$. This quadrant yields a total of 1001 unique minimum paths from source to destination as per Equation 3.1. When considering the number of paths which avoids the periphery, that value is found to be 220 as per Equation 3.4. Analyzing the decision of the next hop in terms of traditional CQR, the x and y are then selected based on the smallest output queue length.

But, this is where Periphery Avoidance can take advantage of having such a difference in the Δ's. Of course, output queues need to be considered, as they are an indicator of network congestion, but routing in the y direction should be preferred as it avoids the periphery better than choosing the x direction for this example.

Therefore, for the enhanced CQR the values obtained from the decision function are:

$$Q_p(x) = (Q(x) + 1) \cdot 0.714$$
$$Q_p(y) = (Q(y) + 1) \cdot 0.285$$

Here, it's demonstrated that if queue sizes are equivalent, the y direction will be selected as the minimum value, and the next hop.

Considering another example which depicts how the decision function factors on deciding a next hop based on a quadrant with $v = \{\Delta x = 4, \Delta y = 4, \Delta z = 0\}$, with equivalent output queues (as with the previous example), the number of paths between source and destination are 70, with 20 paths within the interior of the quadrant.

Therefore the decision function yields:

$$Q_p(x) = (Q(x) + 1) \cdot 0.500$$
$$Q_p(y) = (Q(y) + 1) \cdot 0.500$$

This causes the output queue values to be the sole deciding factor in this scenario.

These examples demonstrate the basic functionality of this decision function and its ability to weight both output queue lengths and routing towards the interior of the quadrant – both of which are important in load balancing and avoiding hotspots within the network.

Chapter 4

The Research Cluster: Thor's Tack Hammer

4.1 Topology

This section describes the topology of the KSU research cluster, Thor's Tack Hammer. The topology of this cluster is one of the characteristics of this parallel system that helps explain its overall performance, and operability.

In an attempt to keep the individual nodes diskless (or operate without the use of individual hard drives), it was necessary to have a single central server connected to each node, providing PXE-booting and NFS file sharing. This server, named Sandlab, provided those services to the nodes of Thor's Tack Hammer, as well as providing the cluster with a firewall from the Internet.

Along with this management network exists the USB interconnection network. This network uses USB patch cables to connect the nodes in the hypercube or torus-mesh topology. During execution of code, the particular traffic demands and routing decisions are done by routing packets within the USB interconnect network, and cannot use the management network. Both networks use separate subnets, the management network using 192.168.0.0/24 and the USB interconnect using 10.0.0.0/16.

Figure 4.1.1 shows a graphical representation of the network topology of Thor's Tack Hammer.

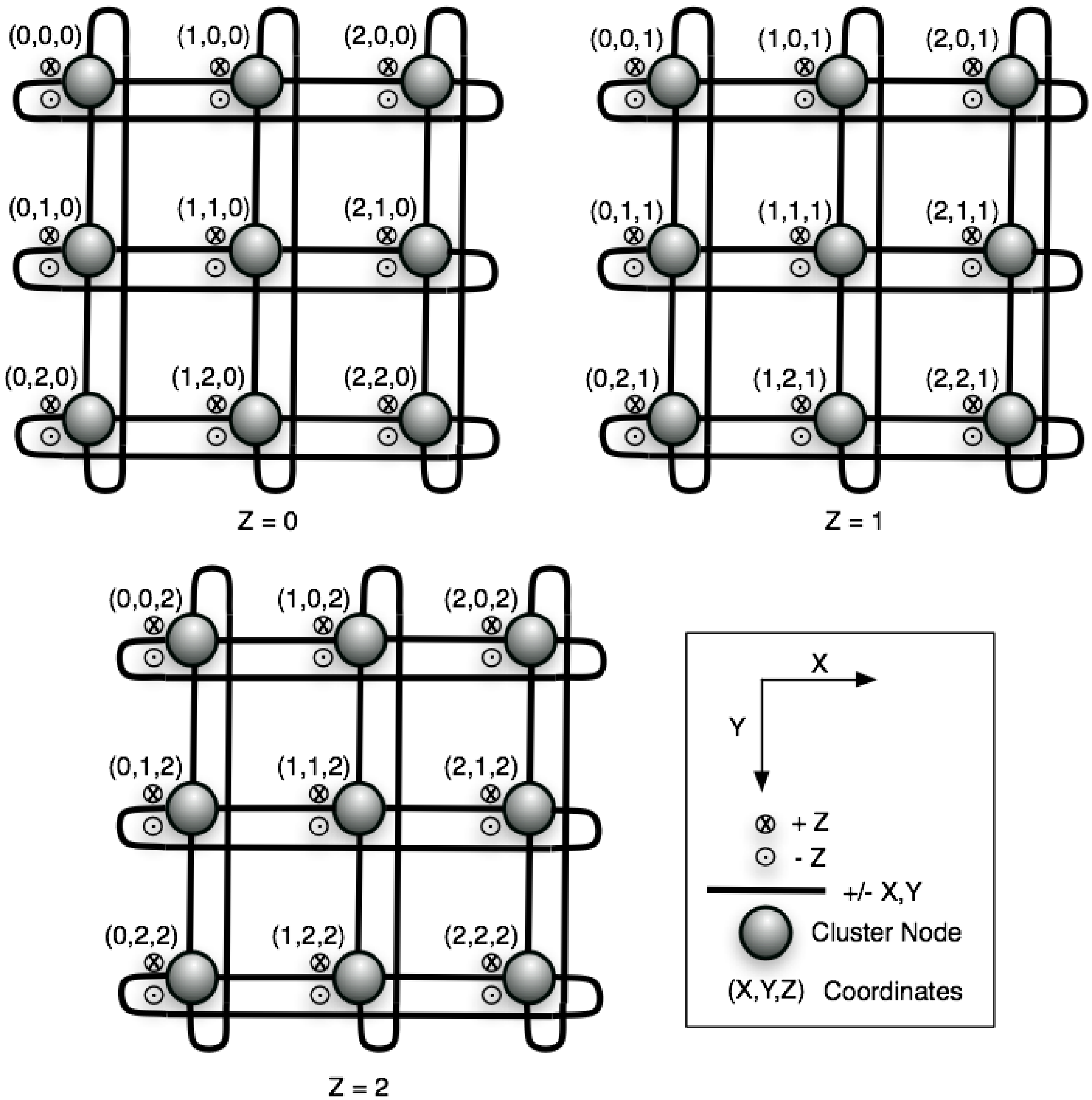


Figure 4.1: *This figure shows the connectivity between nodes in Thor's Tack Hammer.*

4.1.1 Subnetting and Addressing Scheme

Because of the version of Linux that was used, as well as the hardware chosen for the interconnection, each USB interface was able to be used like a traditional ethernet interface. Once the Linux kernel was brought up, the USB interfaces could communicate through the USB patch cables by IP.

This was one of the reasons for choosing these particular patch cables for the interconnect. They were fairly inexpensive (approx. $5-10US per cable), and Linux had drivers for easily transmitting data across the cables.

Subnetting between the nodes was implemented to make communication between a node and its neighbors easy. A subnetting scheme was initially developed by a colleague at Sandia National Laboratories, and is explained below. By creating exclusive subnets for each USB link, it could be asserted that data would be transmitted along the USB interconnect and not the management network. This same scheme was used throughout the research as the standard for connectivity between nodes.

There are first two things to consider when understanding this subnetting scheme. There are slightly different algorithms for calculating a node's local interface addresses and the neighbors' interface addresses. The first part of this discussion explains the calculation of a node's local address (given in x,y,z form - see Figure 4.1).

The first two octets of the addresses can be arbitrarily chosen (x.x.C.D). For Thor's Tack Hammer, '10' and '0' were chosen as the first and second octets respectively. The last two octets are partitioned into 4-bit values, the x-value, the y-value, the z-value, and the direction. Therefore, the third octet contains the x-value and y-value (4 bits + 4 bits), and the fourth octet contains the z-value and the direction (again, 4 bits + 4 bits). Appendix A gives the BASH Linux scripts that were developed, and the code for generating these values is available for reference there. The pseudo-code for calculating a node's the third octet for a positive-x interface is as follows:

$$
\begin{aligned}
\text{C} &= (\text{x-value} << 4) + \text{y-value};\\
& \quad (\textit{alternatively})\\
\text{C} &= (\text{x-value} * 16) + \text{y-value};
\end{aligned}
$$

Whereas, calculating an address for the third octet in a negative-x direction is as follows:

$$
\begin{aligned}
\text{C} &= ((\text{x-value}-1) << 4) + \text{y-value};\\
& \quad (\textit{alternatively})\\
\text{C} &= ((\text{x-value}-1) * 16) + \text{y-value};
\end{aligned}
$$

And calculating the third octet for a negative-y interface:

$$
\begin{aligned}
\text{C} &= (\text{x-value} << 4) + (\text{y-value}-1);\\
& \quad (\textit{alternatively})\\
\text{C} &= (\text{x-value} * 16) + (\text{y-value}-1);
\end{aligned}
$$

Finally for the third octet, the wrap-around must be accounted for since the interconnect is that of a hypercube/torus-mesh. Therefore, a test is done to see if the x-value or y-value is 0. If that is the case, and it is a negative move, the following calculations are done for the negative-x interface:

$$
\begin{aligned}
\text{C} &= ((k-1) << 4) + \text{y-value};\\
& \quad (\textit{alternatively})\\
\text{C} &= ((k-1) * 16) + \text{y-value};
\end{aligned}
$$

And similarly for the negative-y interface:

$$
\begin{aligned}
\text{C} &= (\text{x-value} << 4) + (k-1);\\
& \quad (\textit{alternatively})\\
\text{C} &= (\text{x-value} * 16) + (k-1);
\end{aligned}
$$

The calculations for the fourth octet is similar, but static values for the direction must first be considered. They are as follows:

Positive	X:	dir=1	(0x0001)
Negative	X:	dir=2	(0x0010)
Positive	Y:	dir=5	(0x0101)
Negative	Y:	dir=6	(0x0110)
Positive	Z:	dir=9	(0x1001)
Negative	Z:	dir=10	(0x1010)

These values were chosen to allow for 4 IP addresses per subnet. The values not shown 0 (0x0000), 3 (0x0011), etc., represent the network and broadcast addresses per subnet, which are illegal addresses for the USB interfaces.

Having the appropriate direction values allows for the calculation of the fourth octet. Again, positive and negative directions will alter the way the addresses are calculated (this time with z-value as they were before with x-value and y-value). For positive-z interface the following calculation finds the appropriate octet:

$$\text{D} \quad = \quad (\text{z-value} << 4) + \text{dir};$$

And now for the negative-z interface:

$$\text{D} \quad = \quad ((\text{z-value}-1) << 4) + \text{dir};$$

Finally, when the z-value is equal to 0 and a negative-z direction is necessary the calculation must account for the network's wrap-around:

$$\text{D} \quad = \quad ((k-1) << 4) + \text{dir};$$

In order to allow 4 addresses per subnet as explained above, the interfaces were assigned their 10.0.C.D/30 address and subnet. Calculating the neighbors is done similarly, except the direction values are toggled. Therefore the direction table shown above would change to:

Positive	X:	dir=2	(0x0010)
Negative	X:	dir=1	(0x0001)
Positive	Y:	dir=6	(0x0110)
Negative	Y:	dir=5	(0x0101)
Positive	Z:	dir=10	(0x1010)
Negative	Z:	dir=9	(0x1001)

All other calculations for C and D are done exactly the same as outlined before.

A figure has also been provided giving an example of the results of these calculations. See Figure 4.1.1. This figure shows an entire Z-plane of assigned address, but only for the X and Y interfaces. The Z interfaces were not shown, though they are calculated just as described previously. This example should demonstrate the subnetting and addressing scheme implemented on Thor's Tack Hammer.

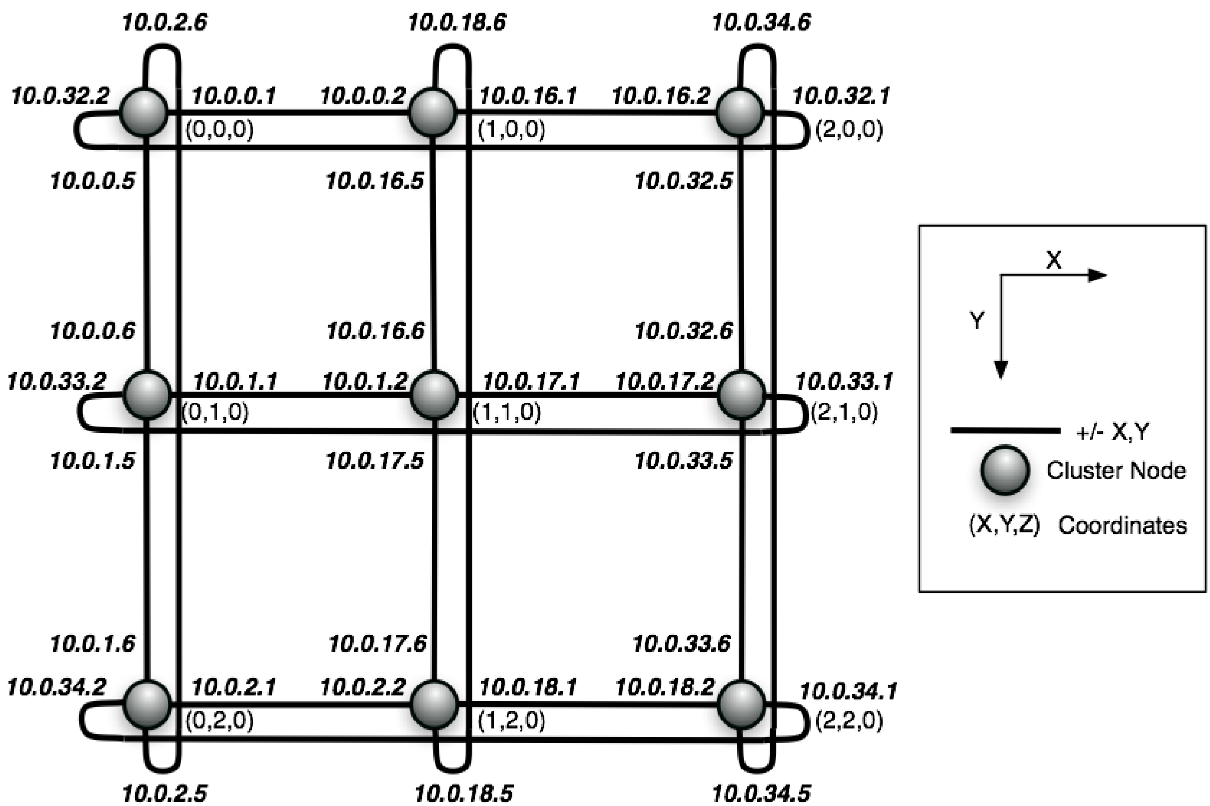


Figure 4.2: *This figure demonstrates the addressing and subnetting scheme used on Thor's Tack Hammer's USB interconnect network. Though the Z interfaces are not shown, they are calculated just as described previously.*

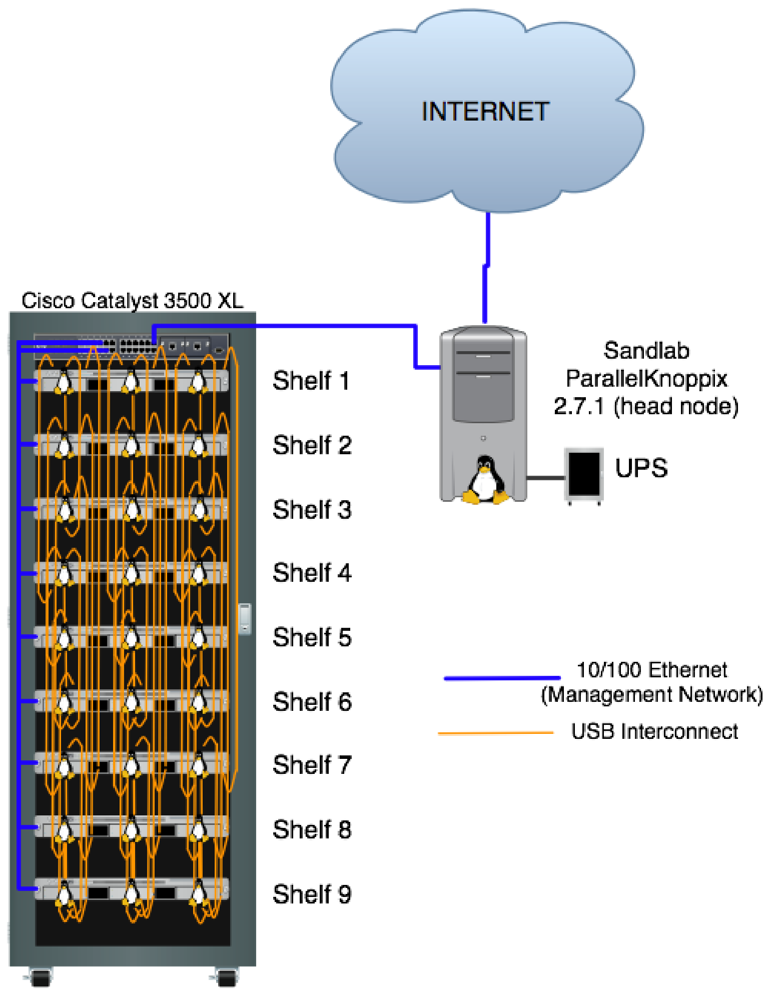


Figure 4.3: *This figure shows the network topology of Thor's Tack Hammer. The management network (shown in blue) is in a star-type topology and the USB interconnect network (shown in orange) is in a hypercube-type topology.*

4.2 Node and Cluster Specifications

Thor's Tack Hammer is composed of 27 nodes, each node being diskless and headless (without individual hard drives and monitors). Each node is a VIA-VT310-DP Mini-ITX (17 cm x 17 cm) Motherboard, complete with the following features:

◇ Dual 1GHz Via Eden^{TM}-N Nano BGA processors
◇ 1GB Kingston KVR400X64C3AK2 DDR RAM
◇ 133 MHz front side bus
◇ Onboard Intel i82551QM 10/100Mbs Ethernet Adapter (used as Management Interface)
◇ Onboard VIA VT6103L 10/100Mbs Ethernet Adapter (unused)
◇ Onboard VIA VT6122 10/100/1000Mbs Ethernet Adapter (unused)
◇ VIA CN400 North Bridge
◇ VT8237R South Bridge
◇ Two onboard USB ports (2.0)
◇ NEC PCI USB card (2.0)
◇ Custom Award BIOS
◇ 27.7 Watts Total Power Consumption

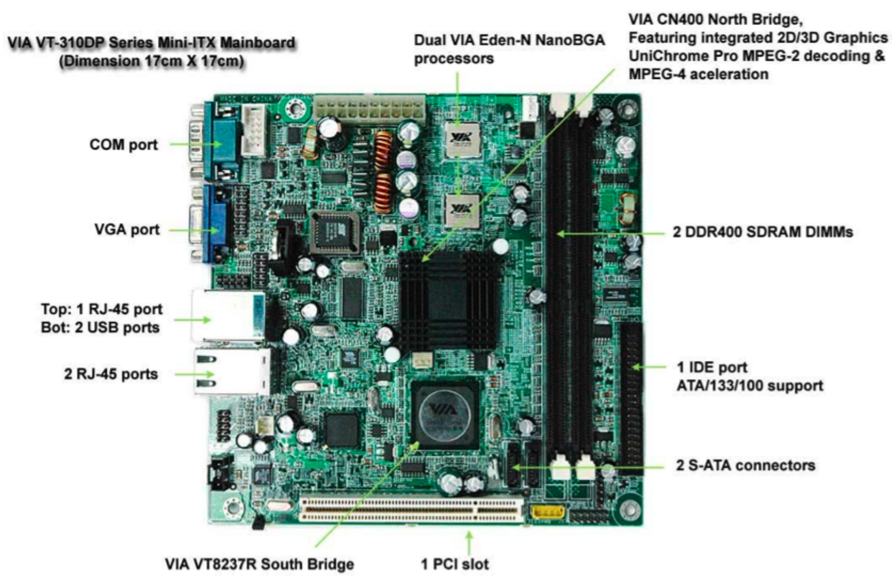


Figure 4.4: *This figure shows the VIA VT310-DP, the motherboard used for each node of Thor's Tack Hammer. (Image borrowed from VIA's site: http://tinyurl.com/2t3lkg.)*

Each node resides with two other nodes (three total) per shelf within the rack, and all three nodes share a common power supply. To enable all nodes to come up simultaneously, wake-on-lan (WOL) was used to switch on the nodes from their off state. Simply turning off the power supply once the nodes have halted was the process for powering down the nodes.

A customized BIOS was needed to allow nodes to PXE-Boot and wake-on-lan on the same interface. This interface ended up being the Intel i82551QM, as opposed to the VIA

Figure 4.5: *This image is a photograph of the Adaptec USB 2.0 AUA-5100 PCI card used to increase the number of USB interfaces on each node. This card used an NEC D720101GJ Chipset. (Image borrowed from http://tinyurl.com/37m8js.)*

VT6122 (Gigabit Ethernet) or the VIA VT6103L interfaces. The engineers at VIA helped develop this customized BIOS for the particular needs of this project.

Coupled with the VIA VT310-DP motherboard was a USB 2.0 PCI card to expand the number of USB slots for the cluster's interconnect network. Using throughput tests, it was found that the NEC USB PCI controller worked better than the others tested. This controller did a good job of balancing the USB bandwidth equally between the connected USB patch cables.

Finally, there were many preliminary operating systems which have been used. Back when this project started in August of 2005, OpenBSD 3.8 and 3.9 were used both on Sandlab (as the PXE-server and NFS server), as well as the diskless nodes. A tutorial for OpenBSD diskless compilation can be found at:

http://www.openbsdsupport.org/diskless.pdf

Soon after, discussion started with the OpenBSD developers towards developing a stable EHCI Ethernet interface via the USB patch cables, as the current version at that date caused kernel panics when attempting to send the maximum amount of data over the USB patch

Figure 4.6: *This image is a photograph of the USB 2.0 NetLink cables used as the interconnect on Thor's Tack Hammer. The links used an ALi M5632 Chipset. (Image borrowed from http://tinyurl.com/37zk7w.)*

cable.

Other operating systems which were considered were PelicanHPC, and CentOS (using OSCAR), but both were discarded after issues with the USB Ethernet drivers and the diskless architecture of Thor's Tack Hammer. Finally, a slightly remastered version of ParallelKnoppix (PK) 2.7.1 was used on the cluster.

The additional packages and tools needed for this particular research was:

1. NTP v.4.2.4p4 (to ensure all nodes have nearly similar clocks)

2. SchedUtils v.1.5.0-1 (CPU affinity)

3. BASH scripts (to automate the cluster's initialization - see Appendix 1)

4. C programs (to generate/route/account for traffic over the interconnect)

The management interface of each network (which was used to PXE-Boot, WOL, and maintain NFS mounts for Linux on each node) was implemented over standard Category-5 Twisted-Pair Ethernet cables. Each cluster node connected to a Cisco Catalyst 3500 series XL switch. The head node also connected to this switch.

4.3 Node and Cluster Performance

There are many possible ways of measuring performance on computational systems and networks. This section is dedicated towards outlining various ways in which Thor's Tack Hammer was benchmarked and ranks against other parallel systems and sequential systems.

Before delving into how Thor's Tack Hammer (or any parallel system) compares to sequential systems, it is important to first consider Amdahl's Law and its application towards understanding parallelization. Amdahl's Law has been used to find the overall expected improvement (or speedup) that can be realized through multiple processors as opposed to a single processor. This law depends greatly on the ratio of which instructions can actually be partitioned onto other processors (with zero communication between them) and computed independently.

Amdahl's Law:	$S_{Total} = \frac{1}{r_s + (\frac{r_p}{n})}$
r_s:	The fraction of instructions that must be run sequentially.
r_p:	The fraction of instructions that can be run in parallel.
n:	The number of processors running the parallel instructions.
S_{Total}:	Total resulting speed-up.

For instance, if one particular task can have 18% of the instructions run in parallel, leaving 82% needing to run concurrently, by executing them across multiple processors, the maximum speedup possible for one processor makes the equation become:

$$1 = \frac{1}{0.82 + (\frac{0.18}{1})}$$

Whereas running the same code on a parallel system with 27 processors, such as Thor's Tack Hammer, we see a maximum possible speedup of:

$$1.209 = \frac{1}{0.82 + (\frac{0.18}{27})}$$

And finally, running that code on BlueGene/L with 212,922 processors, we see a maximum speedup of:

$$1.219 = \frac{1}{0.82 + (\frac{0.18}{212{,}922})}$$

These results clearly depict the necessity of having a good ratio of parallel and sequential instructions in order to truly capitalize on the number of processors in an HPC. With this low ratio of parallel instructions to the sequential instructions, the difference in maximum speedup between Thor's Tack Hammer and BlueGene/L is marginal.

Along with Amdahl's Law, and its impact on the performance of Thor's Tack Hammer, there are other benchmarks available to demonstrate overall performance. These next two sections outline two other benchmarks, throughput between nodes on the USB interconnect, and the Linpack benchmark.

4.3.1 USB Interconnect Throughput

Network throughput between systems can be easily measured by using tools such as Netperf and Iperf. These tools attempt to measure maximum throughput by sending as much data as possible from a source to a destination, given a static amount of time.

In order to measure the throughput between nodes using the USB interconnect, Iperf version 2.0.3 was used. There were two different tests used, first measuring how much data was able to be sent over a single USB link, and the second measuring how much data was able to be sent from a single node to all of its 6 neighbors simultaneously.

The first test attempted to measure the maximum throughput on a single USB link. That maximum throughput was found to be around 76Mbits/sec. The second test disclosed evidence that each node uses two separate USB EHCI (USB 2.0) controllers. The two controllers, a NEC USB controller (via the PCI card, 4 USB ports used), and the VIA USB controller (via the onboard USB ports, 2 USB ports used). Because of the two separate controllers, sending data over all six interfaces totalled at 230Mbits/sec to 265Mbits/sec. This also indicates that the patch cable runs slower than the controller.

4.3.2 Linpack Benchmark

The Linpack benchmark, widely used and accepted as a standard measure of compute power, operates by solving a random dense system of linear equations. The benchmark is the same

System	Number of Processors	Peak GFlops/sec	Top500 Rank	HPL
LLNL-BlueGene/L (US)	212,922	596,378	1^{st}	Y
SNL-Red Storm (US)	26,569	127,531	6^{th}	Y
Thor's Tack Hammer	27	2.299	—	Y
Thor's Tack Hammer	1	0.190	—	N
iBook 1.42Ghz PPC G4	1	0.051	—	N

Table 4.1: *Linpack test results of various machines. These results were obtained from the November 2007 Top500.org website. The left most column indicates whether the Linpack or the Linpack/HPL version was used.*

benchmark used by the Top500.org site to rank the top 500 fastest computers in the world. It uses the BLAS library (Basic Linear Algebra Subprograms) to solve the equations using Gaussian elimination with partial pivoting. The benchmark measures how many millions of floating point operations per second were observed during the computation.

In order to facilitate a proper parallel Linpack benchmark, the Linpack/HPL version of the test tool was downloaded and executed run over MPI (Message Passing Interface).

Various parameters are used to "tune" the benchmark. These parameters vary the ways in which the problem set is partitioned among the nodes within a cluster or larger HPC. The three parameters used to benchmark this cluster are N, which specifies the number of problems to be run; NB, which specifies the block size of the problem set; and finally the P and Q parameters, which partition the entire problem set between the nodes. P and Q are multiplied together, and must be less than or equal to the number of nodes in the cluster.

Documentation that came with the downloaded Linpack/HPC tool explained that by trying all possible combinations of P and Q, the user should find one value that is the best result.

Table 4.1 shows the Linpack test results for various well-known computer systems and HPC systems. Table 4.2 show the varied Linpack/HPL test results for Thor's Tack Hammer during tuning.

Number of Processors	N	NB	P	Q	Time(sec)	Peak GFlops/sec
1	500	100	1	1	438.44	0.190
3	5000	100	1	3	179.95	0.463
27	5000	100	1	27	53.95	1.533
27	5000	100	27	1	76.89	1.099
27	5000	100	9	3	41.10	2.058
27	500	100	3	9	35.78	2.299

Table 4.2: *Varied Linpack/HPL tests for Thor's Tack Hammer after adjusting the tuning parameters.*

Watts	Context During Observation
12	All nodes off; one power supply on; Cisco switch off.
68	All nodes off; all power supplies off; Cisco switch on.
187	All nodes off; all power supplies on; Cisco switch on.
270	3/27 nodes on/idle; all power supplies on; Cisco switch on.
365	6/27 nodes on/idle; all power supplies on; Cisco switch on.
930	27/27 nodes on/idle; all power supplies on; Cisco switch on.
1060	27/27 nodes on/HPL; all power supplies on; Cisco switch on.

Table 4.3: *This table shows the results of power consumption on Thor's Tack Hammer. These results were observed using a Kill-A-Watt P4400 device.*

4.3.3 Power Consumption

This last section demonstrates the power consumption of Thor's Tack Hammer in terms of electricity used. These results were obtained by observing the amount of watts that were being used at various points during the startup and execution of the cluster. The tool used to analyze the watts used was a P3 Kill-A-Watt P4400. The following table shows the various levels of power used and the context in which that observation took place.

Table 4.3 makes a few things clear about the consumption of power for this particular cluster. First, it is interesting to see that because of the low consumption of power per node, this entire cluster uses less electricity than that of a typical hair drier or microwave oven.

It's also worth analyzing further the power consumption per node both with idle and at high utilization, considering all factors included within the network. These factors include power consumption by the switch for the management network and residual power consumption by the power supplies.

Therefore we get values of 34.44 watts per node for idle processing, and 39.26 watts per node for high processor utilization for this particular cluster network. These results should indicate that operating a cluster such as this can be very energy efficient. Also, because of these results, it's also clear that not all parallel systems require complex cooling mechanisms. This cluster maintains a steady temperature only by the fans which circulate air over the processors' heat sinks.

Chapter 5

Laboratory Results

This next chapter explains in depth the methodology for obtaining the results comparing the various algorithms on Thor's Tack Hammer as well as presenting and interpreting those results.

5.1 Methodology for Tests

The method for testing these algorithms within Thor's Tack Hammer was implemented using C code and BASH scripts. The code and scripts that were written were fully commented and are outlined in the appendices.

It was first considered to use routing tables to enable inter-node routing, but upon further consideration, would be difficult if not unreasonable to implement. This is because dynamic implementations would require constant and costly calculations and changes to the routing tables for every packet received.

Along with dynamically changing the routing tables, current transport-layer network daemons which were compatible with dynamic routing algorithms were considered. These included GNU Zebra (www.zebra.org) and Quagga (www.quagga.net). These applications have been under active development for dynamic routing algorithms such as BGP, OSPF, and RIP. This was an option that was considered, but was not implemented.

Therefore, an application-layer socket program was developed to appropriately receive data, calculate the next hop, and forward the data accordingly. A few assumptions were made to verify that this application would in fact be suitable for testing and evaluating the algorithms in question:

Assumption 1: *The use of TCP was necessary to guarantee correct delivery of data between nodes.* This was an assumption made after preliminary evidence that without the flow-control of TCP, UDP could not guarantee correct data delivery. It was also assumed that a layer-4 protocol such as UDP would not be a good implementation choice on a larger supercomputer for this same reason.

Assumption 2: *Time stamps were necessary to compare both the algorithms and the C routing programs.* The time stamps provided crucial information about any particular data's experience traversing through the network. This included actual time spent during the calculations for a data segment's next hop, as well as time spent in transit between nodes. NTP was used to allow a fine resolution of time, and to ensure all nodes had a value nearly exact to that of the other nodes. Because of drifting the estimated error between nodes ranged from 10s to 100s of micro seconds. This is supported in [11].

Assumption 3: *By comparing each routing algorithm's time stamps of service delay, it is possible to compare their complexities.* By using the best-performing algorithm as the baseline, the other algorithms can be compared accordingly. This of course represents a software-based calculation and should not be considered an accurate assessment of a hardware-based implementation.

Assumption 4: *A fork subroutine was necessary to lower the impact of blocking calls onto the processing of other data.* This enabled nodes to continue to process data in parallel over all six USB interfaces simultaneously, and limited the blocking calls to only affect the data associated with that child process.

Assumption 5: *Because access to a node's actual interface output queues was not available at the application level, a shared memory implementation using semaphores would be sufficient.* Output queues were implemented using semaphores and shared memory between the child processes. This was necessary for the fully adaptive algorithms.

Assumption 6: *This C-program implementation aligns with the queueing models of the previous work.* Even though the C-programs implement a queue using forks, and a common area of shared memory protected with semaphores, each interface has a single processor. Also, a constant value was used to decrease the queues when necessary. Therefore this C-program implementation is equivalent to the model proposed in the previous work, which modeled the queues as M/D/1 queues.

These assumptions help justify the way in which validation of the algorithms was performed. They also help explain the reasoning behind the programs' implementations.

5.1.1 A Packet's Progression through the Interconnection Network

It is important to consider how data is not only created, but transmitted, routed, and cataloged throughout the interconnection network using the C programs. This section explains just that.

Step 1: Data Injection. Data is created in chunks of variable size, all "dummy" random data. This data includes a header of fixed size used to contain source, destination, time stamps, and other vital information for proper routing. This all occurs through a program run on the head-node (which is not part of the USB interconnect network) using inject.c, the Injector program. This Injector program reads a file which lists all source/destination traffic demands and paired with the specific routing algorithm, creates the data and sends the data via TCP through the management network directly to the source node. The assumption that all data is injected into the network at nearly the same time can be validated by examining time stamps.

Step 2: Preliminary Routing. The source node then receives the data from the management network and processes the header. This file is the Server file, or server.c. It calculates the next appropriate hop (given the routing algorithm and the destination), and routes accordingly through the USB interconnect network. A time stamp is added at the source node to setup an initial time.

Step 3: Intermediate Routing. The data continues to be received through the USB network and intermediate hops process the data's header and send it onto the next hop using the server.c program. The time that the data spends while being processed (time between receipt of data and transmission to next hop) at each hop is added to the previous service value(s), and the number of hops the packet has traversed is incremented. If this hop count exceeds a predetermined value, it is removed from the interconnect network and processed like it had reached the destination successfully, but flagged as a dropped packet.

Step 4: Destination Routing. Once the data has been received at the destination node, it forwards the header information to another program listening on the loopback interface, loserver.c. This program parses the header once more, and writes valid information to a file which will be parsed by the BASH scripts. A final time stamp is generated at the destination node. CPU affinity allows for this process to run on a completely separate processor than the server.c program.

The server.c program is set to listen on any interface given a specified port, and can only transmit data onto an outbound USB interconnect interface. This enables it to receive data from the management port (during Data Injection), and guarantees it will only transmit the data from that node onto the USB interconnect network. This program runs on all nodes within the interconnect network.

The loserver.c program is set to listen only to the loopback interface on a different port, and receives the header and writes data to files. Some of the data obtained from the header includes average queue times, number of hops the data traversed, and actual time when data was created and removed from the interconnect network. This program runs on all nodes within the interconnect network.

The injector.c program is used to inject data into Thor's Tack Hammer. It is run on only the head-node (Sandlab). It accepts arguments of an input file and the algorithm to route on the USB interconnect network. The input file designates in a matrix form the demand (or number of data segments) to send given every source/destination pair.

5.2 Laboratory Results

This section presents both validation of previous routing algorithms as well as the data gathered from the execution of Enhanced CQR with Periphery Avoidance on Thor's Tack Hammer. All of the assumptions and implementations described in the previous section apply to this data.

The result from the laboratory analysis are presented below by graphically showing their link-load and a histogram of all links normalized. In order to normalize these links, the maximum amount of data transmitted across a single bi-directional link was found and then all others were normalized to that value. Therefore, the highest-utilized link is given a value of 1.0 and all subsequent lower-utilized links have lower values. Those links with zero utilization then have a value of 0 within the histogram.

All algorithms were tested using the Table 2.1 before, and by using the BASH scripts in Appendix A to parse the results. These results were then placed through Matlab scripts listed in Appendix C to graphically view loads and link-utilization distributions.

Static and oblivious algorithms were not tested against the presence of hotspots, as they are unable to make decisions because of them [5]. The adaptive algorithms, however, were compared both with and without the presence of hotspots. See Table 2.1 for an explanation of traffic patterns, and sections (missing ref) for in-depth explanations of each algorithm.

There are some basic indicators of whether or not an algorithm outperforms another. First, a tightly distributed and highly utilized network indicates good load balancing. This may seem counter-intuitive but because the links are all normalized, it is desired. By having all links (or most links) at 100% utilization, that indicates that each link experienced the same traffic load as every other link. This means that for the histograms presented in Appendix E (which demonstrate the distribution of link utilizations), a large grouping near 1.0 or at 1.0 is highly desired. The tight distribution indicates a hight number of links at or near the same value.

Another indicator which proves helpful in comparing the algorithms is the standard

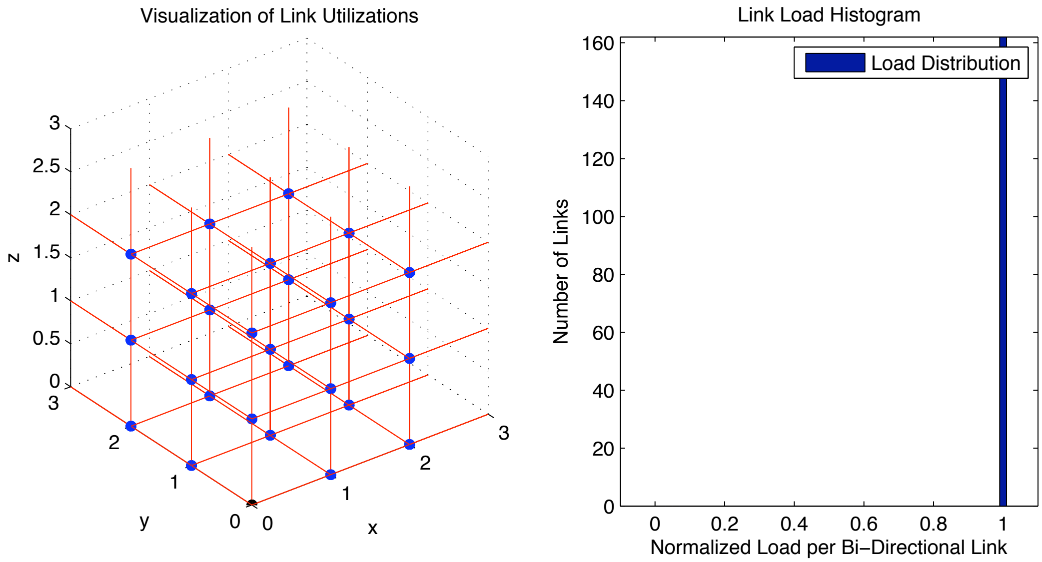


Figure 5.1: *This figure illustrates what are considered excellent laboratory results. Because all links experienced 1.0 utilization after being normalized, they all experienced the exact same traffic load - indicating very efficient load-balancing.*

deviations of the link distributions. Those values are presented in the next few tables. The reason for including these values is to demonstrate how "tight" the link distributions were. The optimum value we search for with standard deviations would be 0.0, indicating all links were at a singular spot. Because of the normalization techniques used, if the standard deviation was 0.0, the utilizations would all be 1.0. This situation is shown in Figure 5.2.

Lastly, by comparing the service times for each algorithm, certain assumptions can be made. In order to demonstrate these scenarios and give examples which indicate good, average, and poor results, Figures 5.2 5.2 and **??** are included below. The first, Figure 5.2 demonstrates a highly utilized network, a zero standard deviation, and illustrates the scenario for absolute optimum load-balancing. Figure 5.2 illustrates an example which is considered good, because of the assumed normal distribution with a fairly low standard deviation. Finally, Figure **??** illustrates bad results, in that many links were at 0.0 utilization (indicating many links were unused during that execution).

On another note, bi-modal distributions could indicate various things, but most certainly,

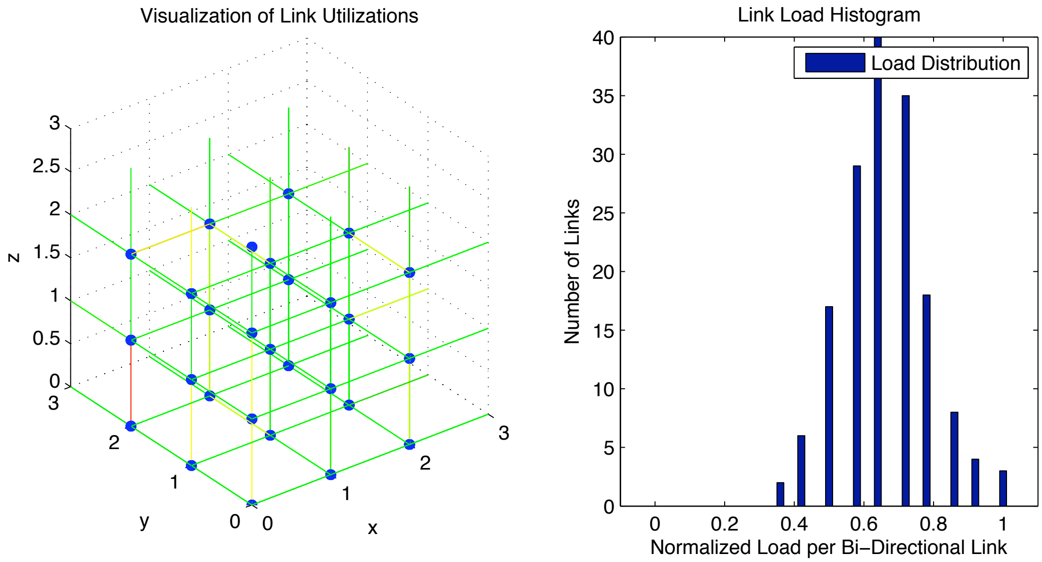


Figure 5.2: *This figure illustrates what are considered OK laboratory results. Because all links experienced traffic loads (none were at 0.0 utilization) and the standard deviation of the link distribution was fairly low, these results were better than poor, but worse than good or excellent.*

indicate faulty decisions given a traffic demand. In certain cases, this bi-modal characteristic is unavoidable (as it is with minimal adaptive given a tornado traffic pattern, for instance).

As it was stated in the assumptions above, this service time can be an indication as to the computational complexity and overhead associated with the implementation of a particular algorithm - both through software and more or less through hardware.

When considering the graph on the left of each result figure (both laboratory and simulation), green lines represent lowly utilized links (which range from $0 : (1 - std\_dev)$, in a normal distribution that is 0-66% utilization), yellow lines represent medium utilized links (which range from $(1 - std\_dev) : (1 - \frac{(std\_dev}{2})$, in a normal distribution, ranges from 66-83%), and red lines representing highly utilized links (which range from $(1 - \frac{std\_dev}{2}) : 1$, in a normal distribution, ranges from 84-100%).

The exact same traffic demands were implemented for these tests as were implemented for the 3-ary 3-cube simulation presented in the previous chapter. Each individual traffic

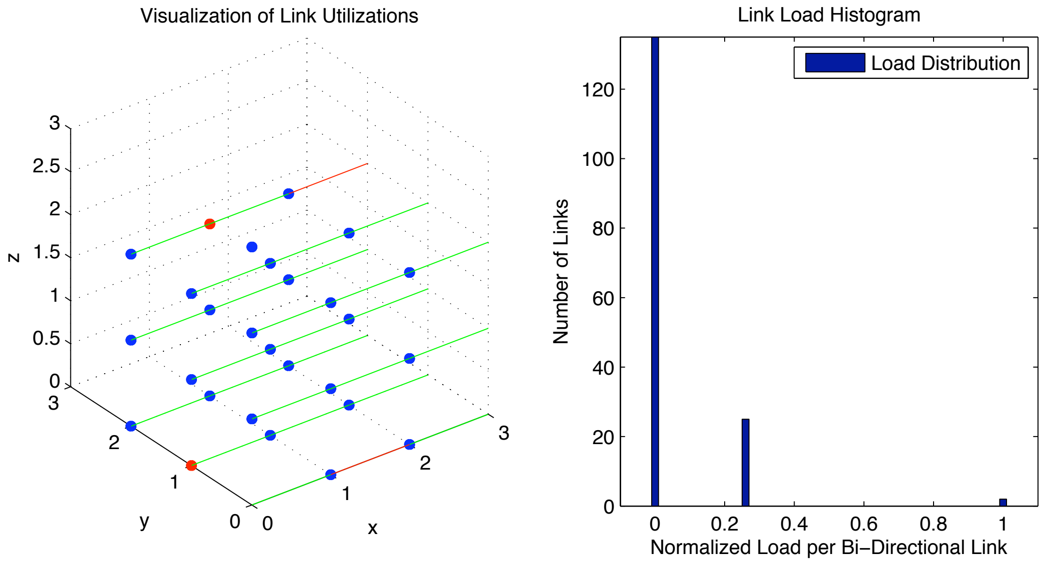


Figure 5.3: *This figure illustrates what are considered poor laboratory results. There were a large number of links which experienced zero traffic (indicating poor load-balancing techniques).*

demand consists of 800KB of data and any TCP overhead associated with its transmission from source to destination. Because of 1500 Byte MTUs on the interconnect, fragmentation and reassembly procedures were necessary. The application layer header was included within the 800KB data. All results presented were the average values after executing the algorithm and traffic demand twice.

5.2.1 Dimension Ordered Routing (DOR) Results

Appendix D.2.1 shows the graphs associated with Dimension Ordered Routing when varying the traffic demands. The benefits of using DOR are visible in what are deemed easy or benign traffic patterns such as NN, UR, and FLOOD. The hard traffic patterns, such as TOR, BC, and TP, did not yield good load-balancing results.

Table 5.1 shows more results obtained during these tests.

Traffic Pattern	NN	UR	BC	TP	TOR	FL
Avg Service Time per Node (ms):	16.6	13.5	14.9	15.0	16.4	13.9
Avg Utilization per Link (%):	64.6	30.6	22.9	26.1	13.5	64.1
Link Utilization Std Dev:	13.8	19.2	33.0	26.9	29.7	7.5

Table 5.1: *This table shows the results obtained from the execution of the various traffic patterns while using Dimension Ordered Routing on Thor's Tack Hammer*

Traffic Pattern	NN	UR	BC	TP	TOR	FL
Avg Service Time per Node (ms):	18.2	13.4	14.6	13.9	15.6	13.8
Avg Utilization per Link (%):	65.5	37.6	8.8	25.6	10.6	73.0
Link Utilization Std Dev:	14.6	23.7	18.2	25.6	23.7	8.6

Table 5.2: *This table shows the results obtained from the execution of the various traffic patterns while using Direction Ordered Routing on Thor's Tack Hammer*

5.2.2 Direction Ordered Routing (DIR) Results

Appendix D.2.2 shows the graphs associated with Direction Ordered Routing when varying the traffic demands. The results of DIR are nearly identical to DOR, and use the same benign and hard traffic patterns as the previous section discusses.

Table 5.2 shows more results obtained during these tests. As the results show, the service times are lower than that of the adaptive and oblivious algorithms, but the utilizations are lower than the others.

5.2.3 Minimal Oblivious Routing Results

See Appendix D.3.1 for the graphs associated with Minimal Oblivious Routing when varying the traffic demands.

Table 5.3 shows more results obtained during these tests.

Traffic Pattern	NN	UR	BC	TP	TOR	FL
Avg Service Time per Node (ms):	19.0	14.0	13.7	14.5	11.6	15.4
Avg Utilization per Link (%):	70.6	43.5	18.3	26.3	11.6	63.3
Link Utilization Std Dev:	18.3	23.9	26.8	27.4	26.0	12.9

Table 5.3: *This table shows the results obtained from the execution of the various traffic patterns while using Minimal Oblivious Routing on Thor's Tack Hammer.*

Traffic Pattern	NN	UR	BC	TP	TOR	FL	FL-HS
Avg Service Time per Node (ms):	21.8	20.6	30.0	28.6	31.3	19.6	23.7
Avg Utilization per Link (%):	60.4	38.9	11.9	23.6	12.4	53.7	33.5
Link Utilization Std Dev:	15.1	20.5	23.5	24.5	27.3	12.6	13.3

Table 5.4: *This table shows the results obtained from the execution of the various traffic patterns while using Minimal Adaptive Routing on Thor's Tack Hammer.*

5.2.4 Minimal Adaptive Routing Results

See Appendix D.4.1 for the graphs associated with Minimal Adaptive Routing when varying the traffic demands. Because hotspots can affect the outcome of the results, the addition of hotspots were included with the FLOOD traffic pattern

Table 5.4 shows more results obtained during these tests. The results shown here indicate that the implementation of an adaptive algorithm increases the service time, but they also show that the utilization increased because of the ability to make better routing decisions and distribute the load better. This data aligns with that of the previous work in [13].

5.2.5 CQR Routing Results

See Appendix D.4.2 for the graphs associated with CQR Routing when varying the traffic demands. Because hotspots can affect the outcome of the results, the addition of hotspots were included with the FLOOD traffic pattern

Table 5.5 shows more results obtained during these tests. This algorithm showed a slight improvement over the utilizations from Minimal Adaptive. CQR also had similar service times, as should be expected. This data also aligned with the previous work of [13].

Traffic Pattern	NN	UR	BC	TP	TOR	FL	FL-HS
Avg Service Time per Node (ms):	24.3	20.5	26.5	22.4	32.4	23.0	22.5
Avg Utilization per Link (%):	46.8	38.8	12.1	29.9	14.3	45.4	33.1
Link Utilization Std Dev:	25.7	17.9	17.1	22.5	26.0	17.9	16.2

Table 5.5: *This table shows the results obtained from the execution of the various traffic patterns while using CQR Routing on Thor's Tack Hammer.*

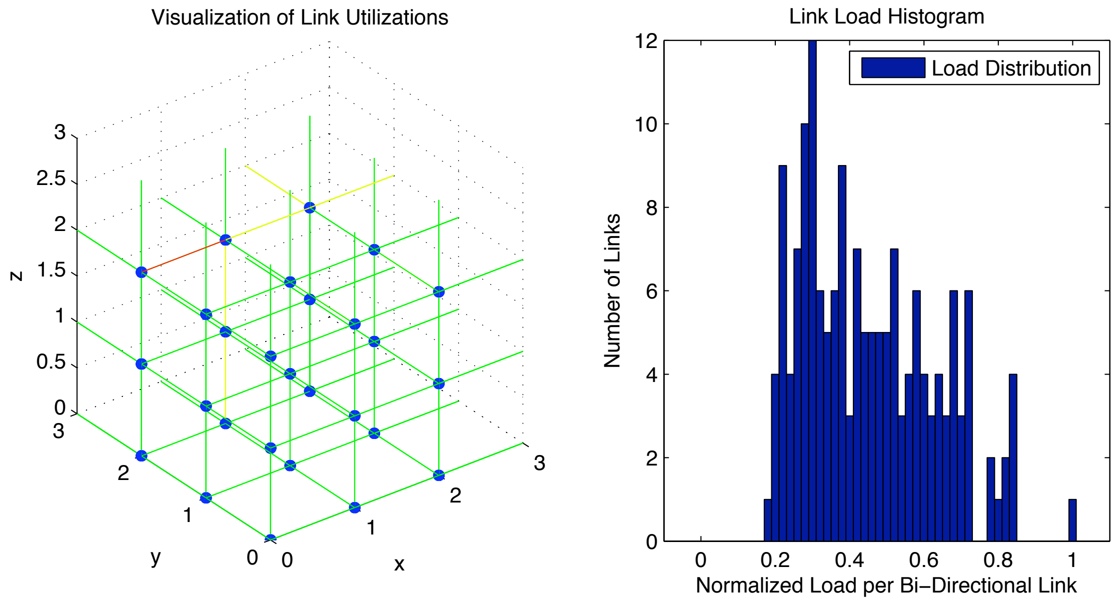


Figure 5.4: *Results of CQR Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

Also, it is worth noting that both algorithms, Minimal Adaptive and CQR, showed a heavy decrease in the average utilization when encountering hotspots. Both responded similarly to the hotspot traffic, dropping the average nearly 20%.

5.2.6 Enhanced CQR Routing Results

See Appendix D.4.3 for the graphs associated with Enhanced CQR Routing when varying the traffic demands. Because hotspots can affect the outcome of the results, the addition of hotspots were included with the FLOOD traffic pattern

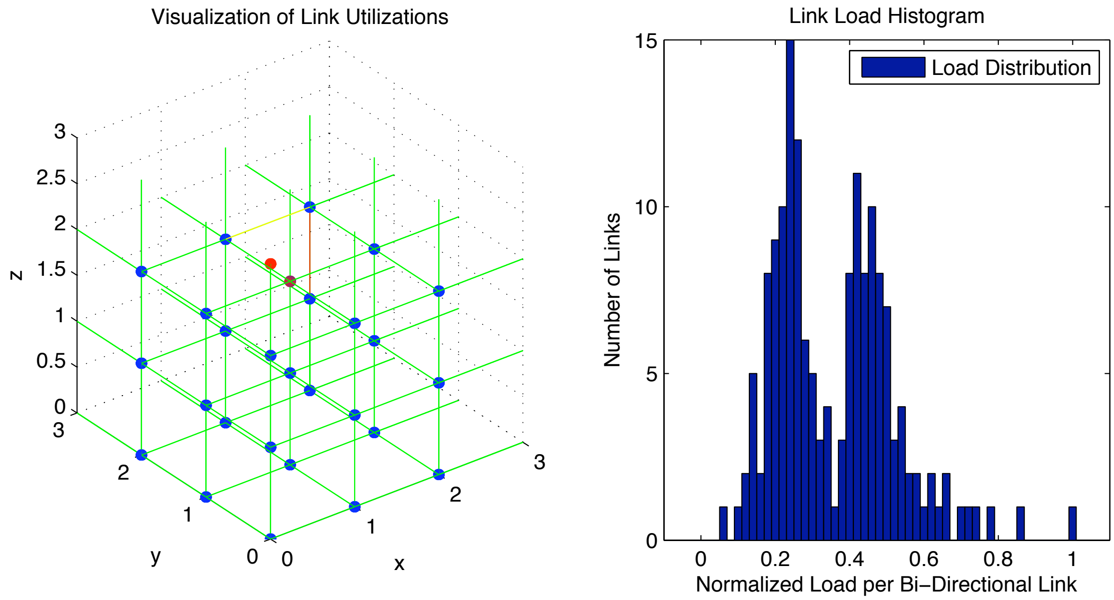


Figure 5.5: *Results of CQR Routing using a flood traffic pattern (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

Traffic Pattern	NN	UR	BC	TP	TOR	FL	FL-HS
Avg Service Time per Node (ms):	23.7	22.6	27.7	21.1	35.1	21.7	16.5
Avg Utilization per Link (%):	45.1	43.4	14.4	32.6	16.4	49.2	41.4
Link Utilization Std Dev:	23.2	19.0	21.3	22.6	27.4	19.8	20.0

Table 5.6: *This table shows the results obtained from the execution of the various traffic patterns while using Enhanced CQR Routing on Thor's Tack Hammer.*

Table **??** shows more results obtained during these tests. These results indicate that this enhanced version of CQR responds very similarly to that of traditional CQR in each tested traffic pattern. As for the FLOOD traffic pattern with hotspots, Enhanced CQR with Periphery Avoidance shows great potential, as it exceeds traditional CQR's results with a 10% higher average utilization. These results do show that in the presence of hotspots, Enhanced CQR with its modified decision function, gives better results when encountering hotspots within a network.

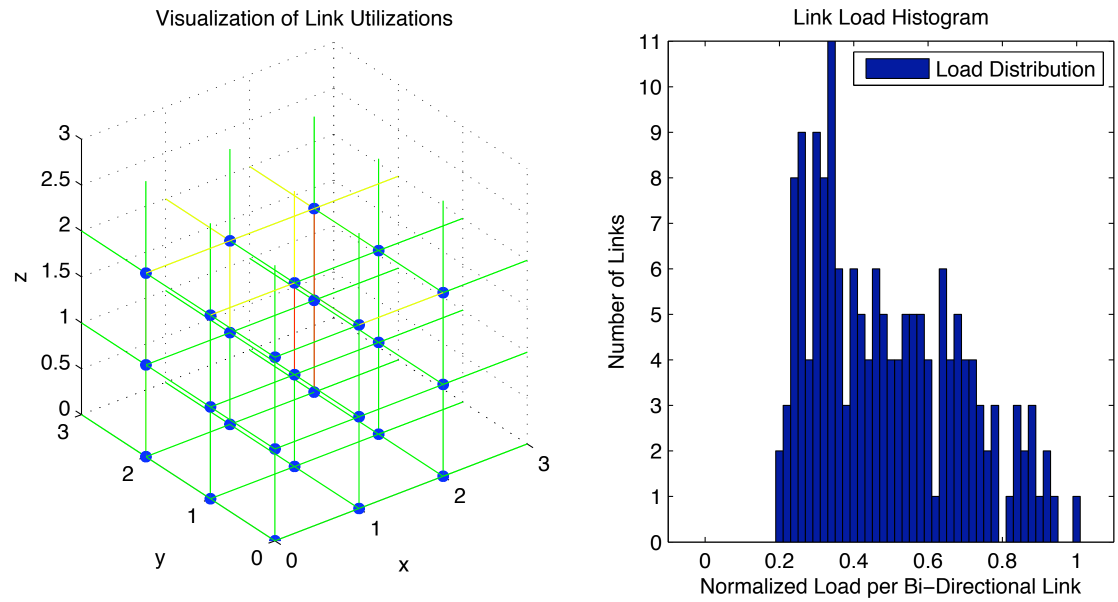


Figure 5.6: *Results of Enhanced CQR Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

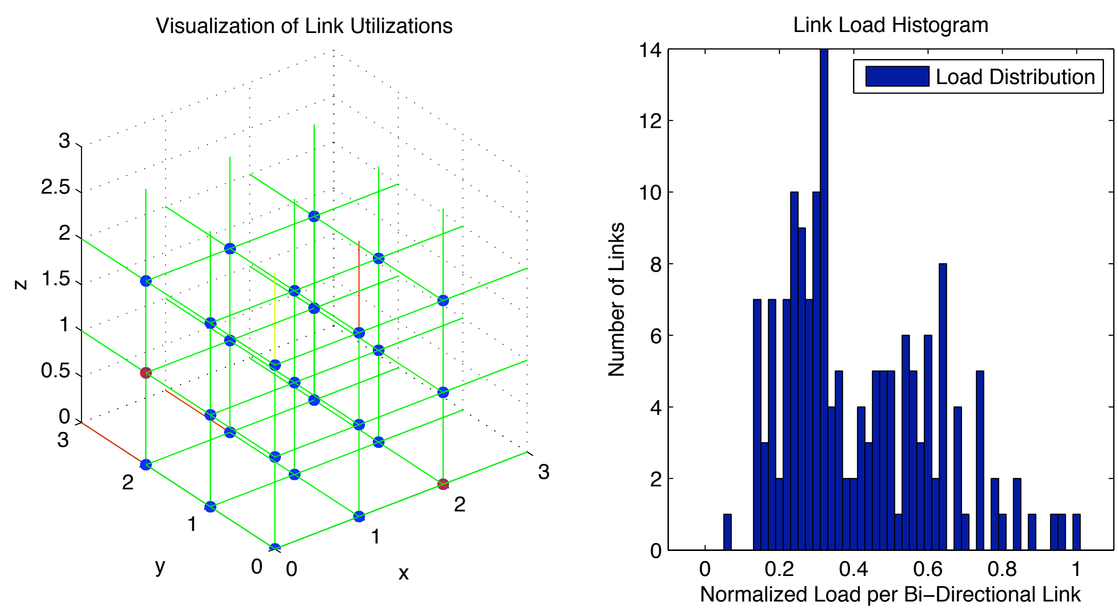


Figure 5.7: *Results of Enhanced CQR Routing using a flood traffic pattern (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

5.2.7 Comparison of All Results

In order to easily compare all of the results presented above, an elementary metric was developed. Two of the values obtained from results have an inverse relationship based on the following:

1. Good simulation results have high average utilizations. Because these values are all normalized, high utilizations indicate good load-balancing occurred. On the other hand, low utilizations indicate heavy-tails on the link utilization distribution, meaning a few links were very highly utilized compared to the links with average utilizations.

2. Good simulation results have low service times. As algorithms become more complex, their service times will increase. This is because it becomes more complex to calculate a next hop along the path. Therefore as the service times increase, the algorithm becomes more costly to implement.

The metric that was developed to compare the results uses this inverted relationship between link utilizations and service times to give a very elementary indication of overall performance. This metric does not weight the utilizations and service times, but that would not be difficult to include in future work.

$$m = \frac{Utilization}{Service\_Time} \tag{5.1}$$

Where

$Utilization$ is the normalized utilization between 0.0 and 1.0,

$Service\_Time$ is the service time measured in milliseconds.

As Table 5.7 shows, DOR and DIR algorithms performed very similarly, which was expected. For all algorithms, the hard traffic patterns gave much lower values, while the benign pat-

Routing Algorithm	NN	UR	BC	TP	TOR	FL	FL-HS
DOR	3.89	2.27	1.54	1.74	0.82	4.61	–
DIR	3.59	2.80	0.60	1.84	0.68	5.29	–
MO	3.72	3.11	1.34	1.81	0.10	4.11	–
MA	2.77	1.88	0.84	0.82	0.40	2.74	1.41
CQR	1.93	1.89	0.46	1.33	0.44	1.97	1.47
ECQR	1.90	1.92	0.52	1.54	0.47	2.27	2.50

Table 5.7: *By using the metric explained within Equation 5.1, it was possible to do an elementary comparison between all laboratory results. These results should only be used to demonstrate some level of increased performance when the routing algorithms gave higher values.*

terns gave higher values. CQR and ECQR also performed very similarly, except in the presence of hotspots, which ECQR showed increased performance over CQR.

Chapter 6

Simulation Results

This next chapter explains in depth the methodology for obtaining the results comparing the various algorithms by using Matlab scripts.

6.1 Methodology for Tests

The method for creating and testing the various routing algorithms on different topologies was made possible by creating a simulation environment within Matlab. The beginnings of this environment was initially developed by Dr. Don Gruenbacher, but was heavily modified to meet the needs of this research work.

By utilizing the Matlab functionality of matrices, it was possible to easily implement a simulation which would test an algorithm's ability to balance load, and minimize delay between source and destination given a specific topology.

The process by which this simulation environment operates is outlined in a numerical form below:

1. The user inputs simulation parameters such as the specific traffic demand to simulate, the routing algorithm to implement, k and n values for the topology, and probabilities for hotspots and/or nodal failures.

2. Given those values, the script first generates a traffic-demand matrix, which is k^nxk^n in size, using rows as sources and columns as destinations. Hotspots are included

within the calculation of this matrix.

3. Depending on the routing algorithm that was selected, the traffic matrix is then passed to its corresponding function, which returns a load matrix and delay matrix. The load matrix is also of size k^nxk^n, but only has values within (i, j) where i and j are neighbors and sent or received data during the simulation. The delay matrix is of size 1xk^n, and includes the value for each node regarding how many packets it had to route non-minimally, enqueue beyond another time step, or drop (depending on the routing algorithm) – each of these possibilities represent some form of delay.

4. For the static algorithms, paths are calculated and simply added to the global traffic matrix. That matrix is then normalized by finding the highest utilized link and dividing the rest by that value. The same is done for oblivious algorithms. For adaptive algorithms, a different approach was necessary. Each simulation for the adaptive algorithms included individual time-steps, progressing every data segment one single hop, and then doing so until each segment successfully reaches its destination. This particular methodology gives way for the data to make adaptive routing decisions as it progresses through the network. Simply calculating a full path and adding it to the resulting traffic matrix would not provide an accurate model for this type of algorithm. The traffic matrix for adaptive algorithms are also normalized as they were for static algorithms.

5. Once the traffic matrix for a simulation has been calculated, it is then processed through another script which displays the link utilizations in a graphical form, and specific to the topology. Coupled with that display is a histogram depicting the distribution of the link utilizations.

Given the simulation environment described above and the code given in Appendix C, three topologies were fully examined. First, a 3-ary 3-cube was examined, which would coincide with the results from Thor's Tack Hammer (also a 3-ary 3-cube topology). Sec-

Routing Algorithm	NN	UR	BC	TP	TOR	FL	FL-HS
DOR	100%	34.1	33.0	26.7	16.7	100	–
DIR	100%	33.2	8.3	33.3	16.7	100	–
MO	100%	31.7	11.1	26.7	16.7	60.0	–
MA	100%	49.9	16.7	44.4	16.7	75.0	45.8
CQR	39.4%	46.8	19.4	41.8	16.7	65.9	35.2
ECQR	39.4%	52.6	18.4	46.6	16.7	66.2	43.6

Table 6.1: *This table demonstrates the average utilizations for each routing algorithm given the 3-ary 3-cube topology of the simulation environment.*

Routing Algorithm	NN	UR	BC	TP	TOR	FL	FL-HS
DOR	0.0	21.2	47.0	27.8	37.7	0.0	–
DIR	0.0	20.2	16.7	32.0	37.8	0.0	–
MO	0.0	22.9	20.4	27.7	37.4	14.8	–
MA	0.0	17.6	25.4	26.5	37.3	9.11	13.8
CQR	30.4	17.7	23.8	16.5	37.4	12.5	12.0
ECQR	30.3	17.2	20.1	20.4	37.4	13.8	13.5

Table 6.2: *This table demonstrates the standard deviations for each routing algorithm given the 3-ary 3-cube topology of the simulation environment.*

ondly, a larger network of 4-ary 3-cube was simulated, while lastly a 5-ary 3-cube was also simulated. Those results are provided in the next few sections.

6.2 Results from the 3-ary 3-cube Simulations

Tables 6.1 and 6.2 show all results obtained from the simulations in a 3-ary 3-cube hypercube topology. All tests used the exact traffic demands that were used within the 3-ary 3-cube laboratory results of the previous chapter, also shown in Appendix D.

The first table shows the average utilizations, and the second table shows the standard deviation for that specific simulation. The graphs of these results are available in Appendix E.1.

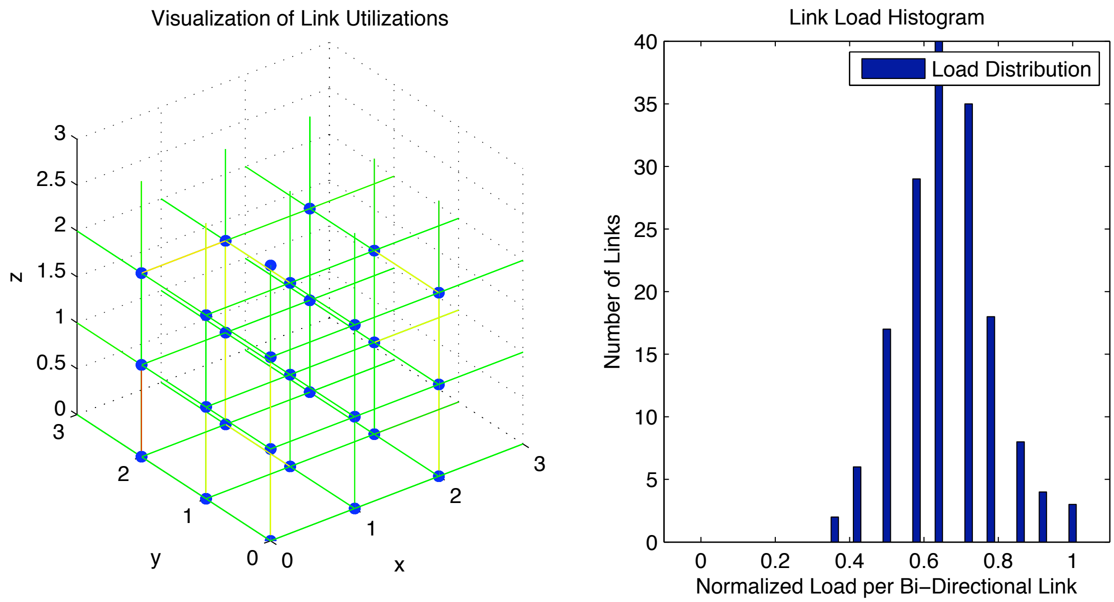


Figure 6.1: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

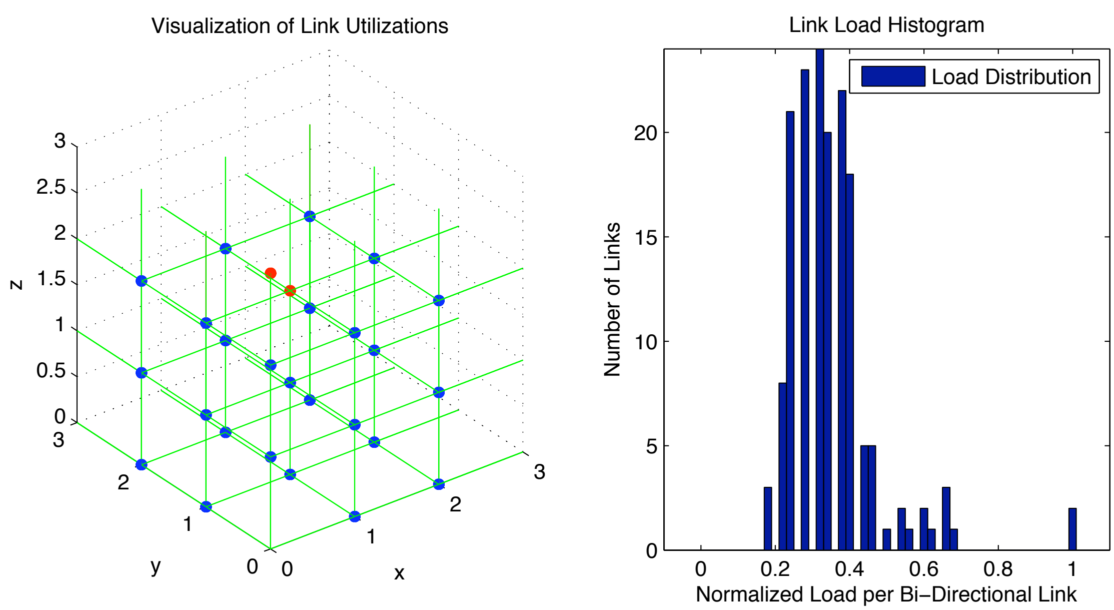


Figure 6.2: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

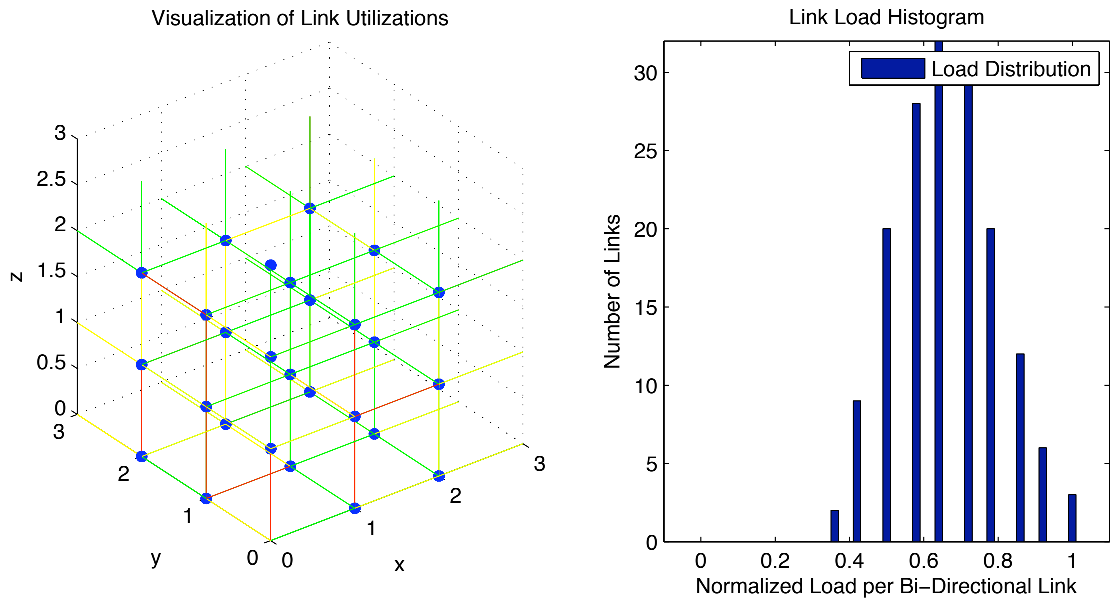


Figure 6.3: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

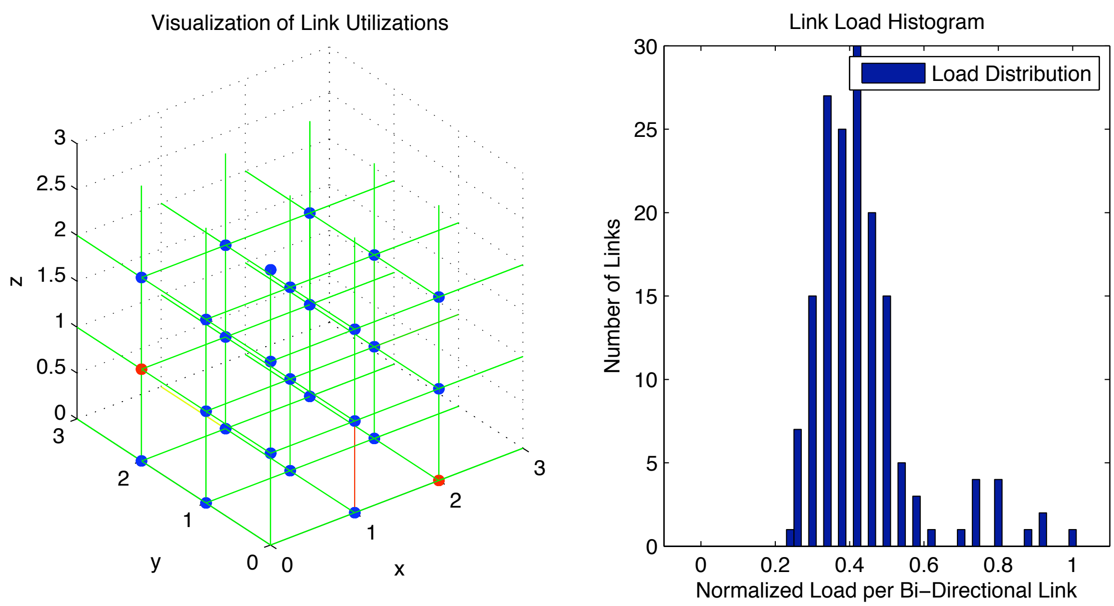


Figure 6.4: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

Routing Algorithm	NN	UR	BC	TP	TOR	FL	FL-HS
DOR	100%	30.1	50.0	25.0	16.7	80.0	–
DIR	100%	32.8	12.5	25.0	16.7	80.0	–
MO	100%	31.0	16.7	22.2	16.7	59.0	–
MA	100%	58.4	25.0	44.4	16.7	85.3	50.3
CQR	34.2%	56.6	25.0	47.6	16.7	75.6	45.9
ECQR	34.4%	51.2	35.0	46.0	16.7	69.5	41.1

Table 6.3: *This table demonstrates the average utilizations for each routing algorithm given the 4-ary 3-cube topology of the simulation environment.*

Routing Algorithm	NN	UR	BC	TP	TOR	FL	FL-HS
DOR	0.0	17.5	50.0	25.4	37.3	6.4	–
DIR	0.0	16.9	23.4	21.2	37.3	7.8	–
MO	0.0	18.5	26.7	18.5	37.3	16.9	–
MA	0.0	13.4	29.5	18.3	37.3	5.4	10.4
CQR	21.5	12.8	23.5	16.5	37.3	8.01	10.2
ECQR	31.5	14.8	23.5	17.8	37.3	15.1	12.2

Table 6.4: *This table demonstrates the standard deviations for each routing algorithm given the 4-ary 3-cube topology of the simulation environment.*

6.3 Results from the 4-ary 3-cube Simulations

Tables 6.3 and 6.4 show all results obtained from the simulations in a 4-ary 3-cube hypercube topology. The first table shows the average utilizations, and the second table shows the standard deviation for that specific simulation. The graphs of these results are available in Appendix E.2.

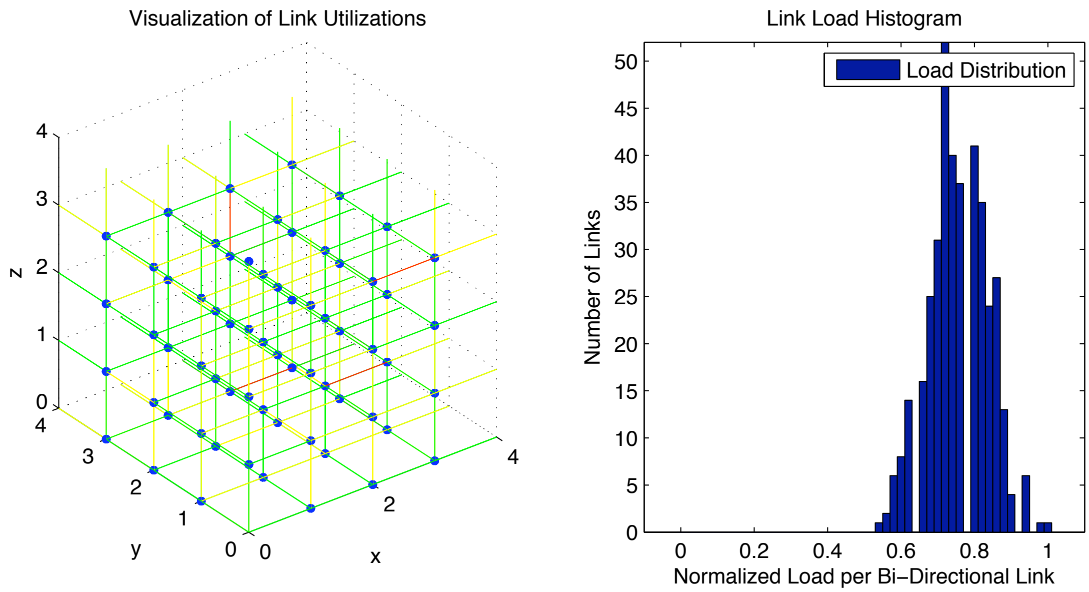


Figure 6.5: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

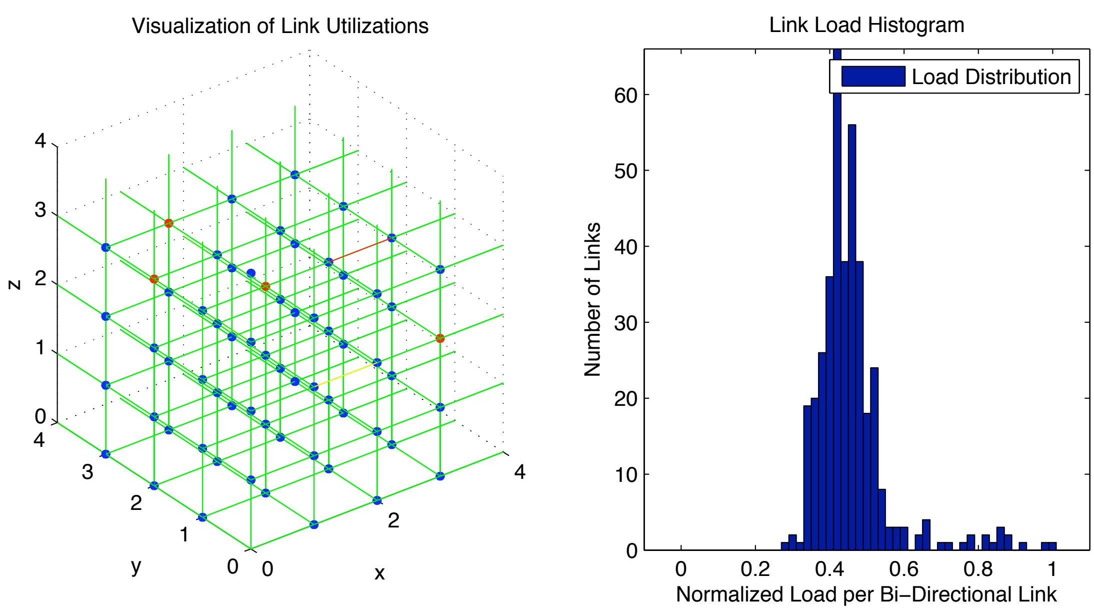


Figure 6.6: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

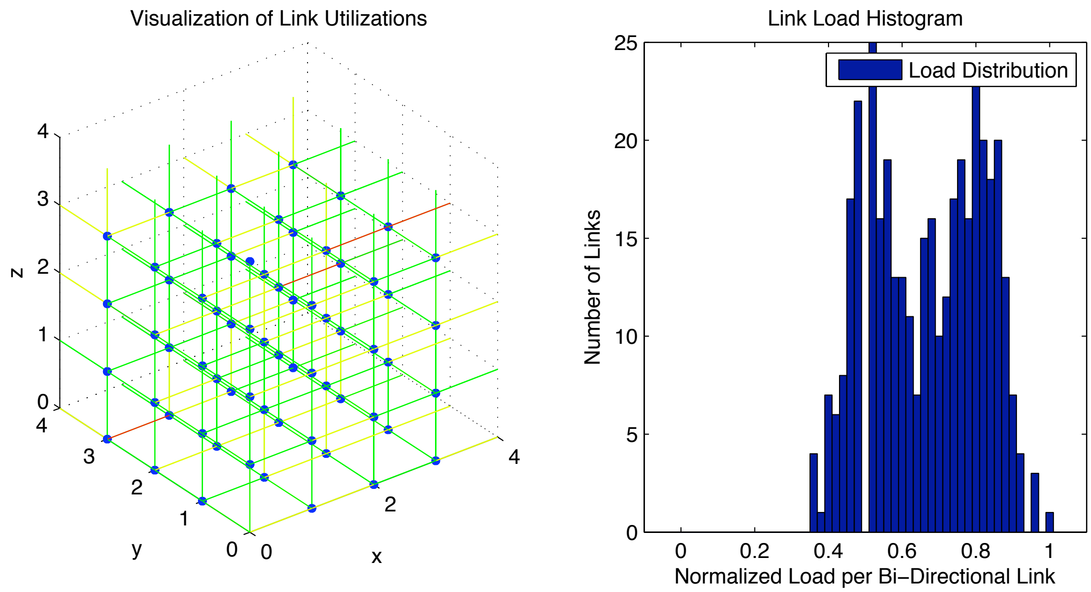


Figure 6.7: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

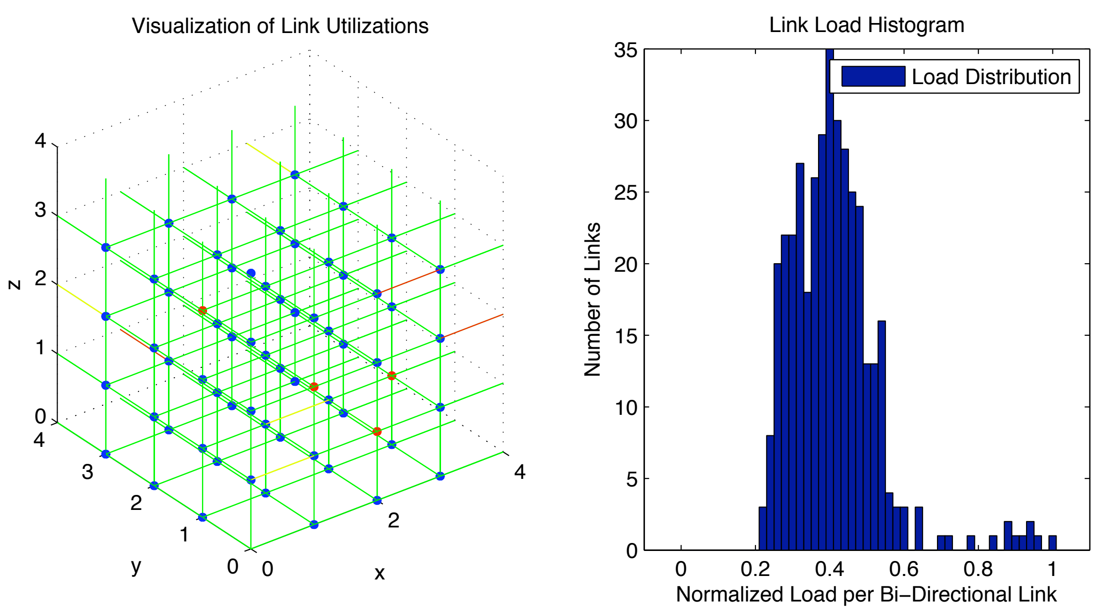


Figure 6.8: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

Routing Algorithm	FL	FL-HS
MA	87.2%	53.7
CQR	86.0%	51.0
ECQR	81.8%	49.1

Table 6.5: *This table demonstrates the average utilizations for each routing algorithm given the 5-ary 3-cube topology of the simulation environment.*

Routing Algorithm	FL	FL-HS
MA	4.5	9.0
CQR	4.6	9.3
ECQR	5.2	9.3

Table 6.6: *This table demonstrates the standard deviations for each routing algorithm given the 5-ary 3-cube topology of the simulation environment.*

6.4 Results from the 5-ary 3-cube Simulations

Tables 6.5 and 6.6 show all results obtained from the simulations in a 5-ary 3-cube hypercube topology. The first table shows the average utilizations, and the second table shows the standard deviation for that specific simulation. The graphs of these results are available in Appendix E.3.

These tables are much smaller than the previous two topologies, solely because of the large execution times involved in the simulations of this network size. Because of this, only the adaptive algorithms are shown, along with the FL and FL-HS traffic patterns. All others have been excluded.

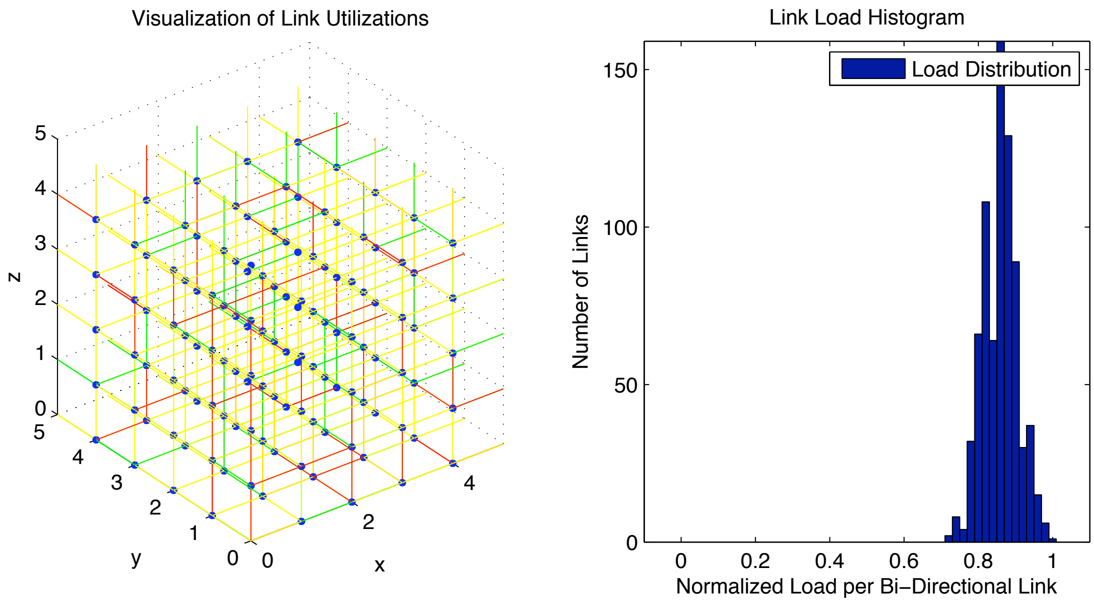


Figure 6.9: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

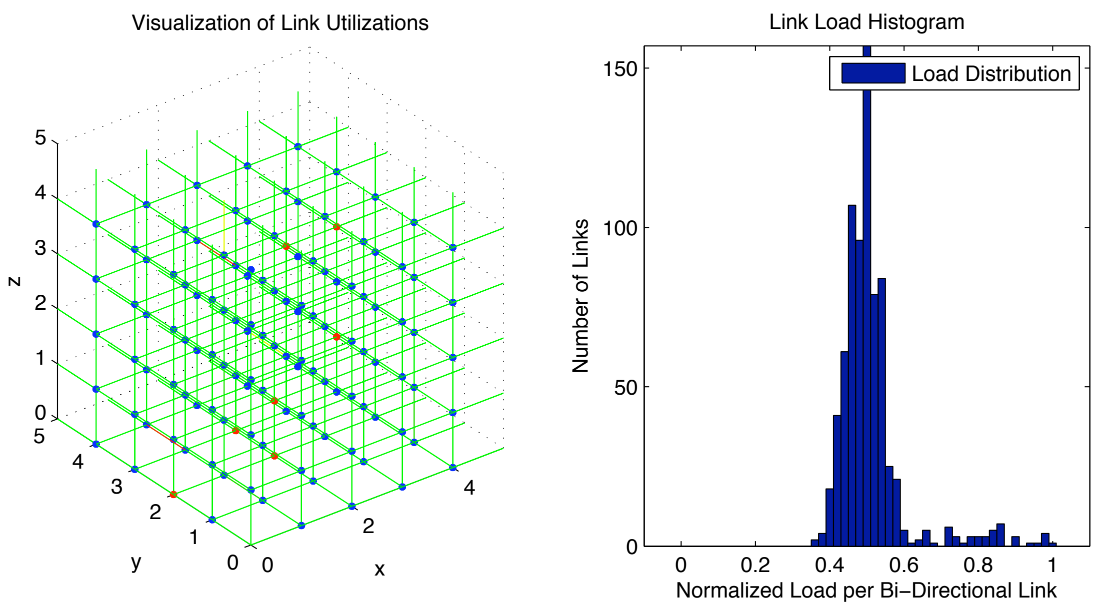


Figure 6.10: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

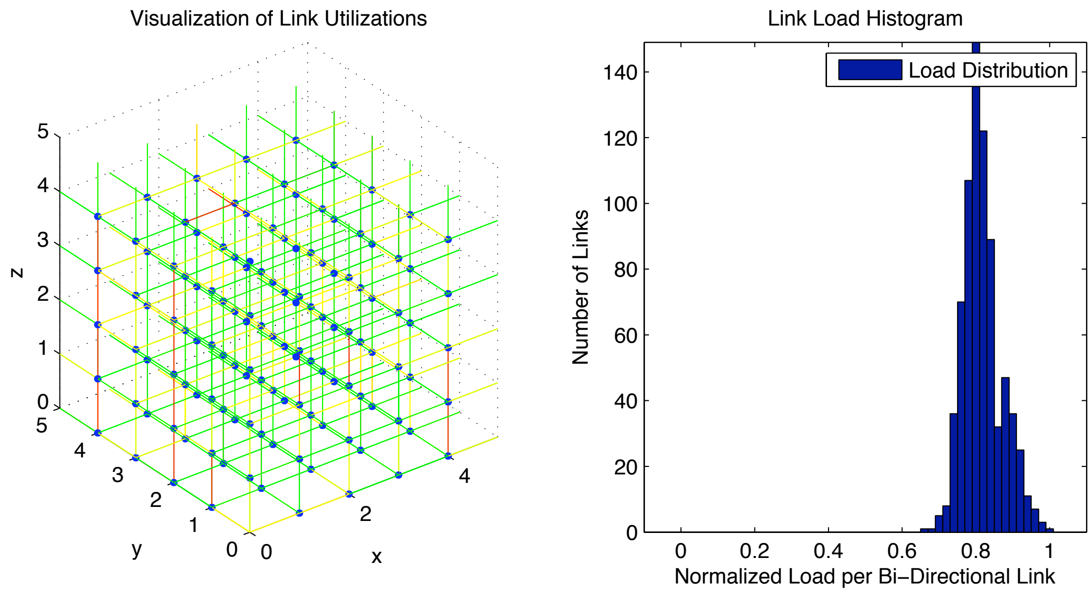


Figure 6.11: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

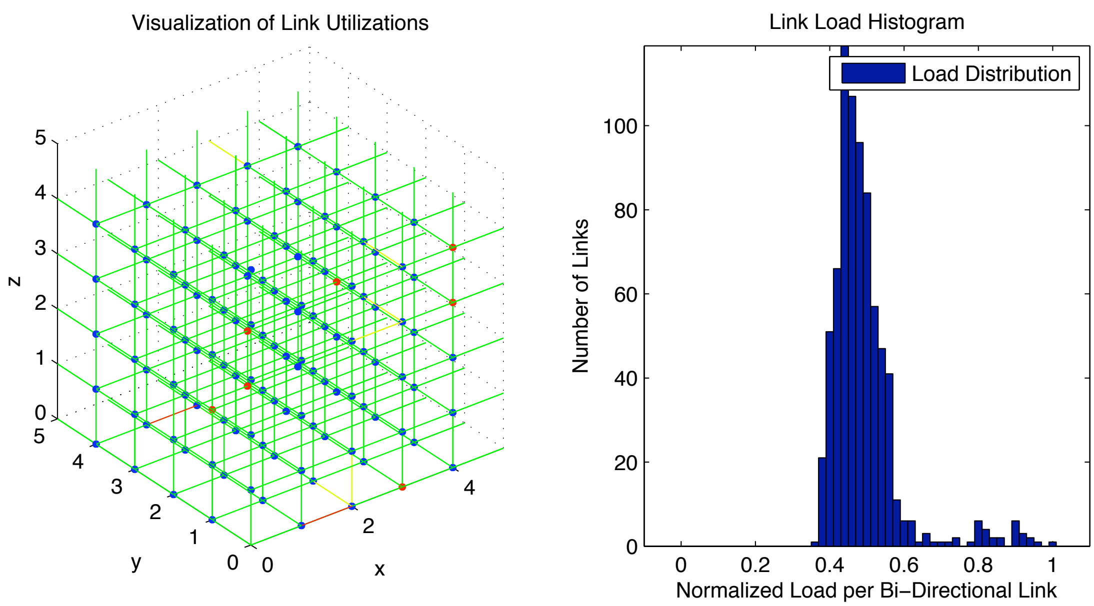


Figure 6.12: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

6.4.1 Comparison of All Results

Taking into consideration the laboratory results obtained within Chapter 5, the results obtained from these simulations should indicate a more-or-less best-case representation of the algorithms. Within the simulation environment the various OSI layers have been ignored, while they definitely had impact on the laboratory experiments. For instance, the layer 3 functionality such as window-sizing and segmentation were not considered within these simulations.

It is visible in the results that there is balancing game that occurs between ensuring locality and efficient load-balancing between the fully-adaptive algorithms. Because of Minimal Adaptive's inability to route non-minimally, its Nearest Neighbor (NN) results showed 100% utilization, while CQR and ECQR have to make decisions as to when to route non-minimally in order to balance the load more efficiently. The values which cause the change-over between minimal/non-minimal between CQR/ECQR should be further analyzed, as the efficiency could possibly increase (thus, increasing the average utilization).

It's also worth pointing out some scaling issues which may be occurring. The results have indicated that with the 3-ary 3-cube topology, ECQR increased performance exceeded that of CQR. As the network became larger with the 4-ary 3-cube and 5-ary 3-cube topologies, this difference shifted. This could indicate that the enhancements made within this work do not scale (or aren't shown to scale here), or that there exists a certain scenario or specific ratio of hotspots to non-hotspots, which depict this increased performance.

The bit complement (BC) traffic pattern has an interesting affect on the Direction Routing Algorithm (DIR), in that its level of performance is lower than that of Dimension Ordered Routing (DOR). These oddities are confirmed on the various topologies, as well as through the laboratory experiments.

Finally, the tornado (TOR) traffic pattern has been shown (both here, and in previous work) to be considerably difficult to respond to. Neither of these algorithms outperformed another when implementing this traffic pattern, as was expected.

Chapter 7

Conclusions

As interconnect networks become more prevalent within electrical and computer engineering, research such as this will continue to help drive towards more efficient routing implementations. Routing decisions, which is the main topic of discussion here, is only one contribution towards higher efficient and more productive data transfer between systems. Whether these systems are cores of a multi-core processor, or nodes within a HPC system, the method for data exchange between the individual systems is essential to increasing performance.

This research thesis contributed to this efficiency by:

- Implementing a simulation-based environment within Matlab to compare the various algorithms' implementations within hypercube topologies;
- Building a research cluster: a scaled version of the Red Storm/Thor's Hammer HPC;
- Implementing a torus-mesh network on this research cluster using a USB interconnect network;
- Implementing various static, oblivious, and adaptive routing algorithms using a set of socket-layer C programs, validating and verifying previous work, and demonstrating this work's implementation differences;
- Introducing Enhanced CQR with Periphery Avoidance, which makes intermediate routing decisions by routing productively but avoiding the perimeter when possible;

- Discussing the impact of this new enhancement and demonstrate evidence of its potential.

The Matlab simulation environment which was developed for this research work, and came from a basic implementation courtesy of Dr. Don Gruenbacher, was thoroughly extended from that early state to fully implement a broad set of routing algorithms and network traffic demands. All extensions which were necessary to this work include:

- Adding a user-friendly GUI, enabling user-input values to quickly expedite the simulation;
- Extending the routing capabilities from only static and oblivious routing to adaptive and fully-adaptive routing simulation capabilities;
- Adding capabilities for generating hotspots and nodal failures during simulations; and
- Adding analysis metrics such as usage, and delay for further analysis.

Singularly using only the Matlab scripts or the laboratory results from Thor's Tack Hammer may not have been sufficient for a high level of confidence in reporting the results of this thesis. Therefore both were invaluable in the analysis of both previous and new routing algorithms.

The research cluster was built from scratch, all commercially available components, and was shown to be an efficient way to implement a small parallel system. The power consumed during execution also suggested that this particular cluster could be very attractive to those who wish to dip their toes into the pool of high performance computing clusters, while not spending too much in implementation and maintenance. Also because of its low power consumption, no extra means for temperature control were necessary (as they would be in most cases).

The results which came from the Matlab simulations and the laboratory results indicated that the concept of periphery avoidance within routable quadrants was one which enabled better hotspot avoidance. This was clear for the 3-ary 3-cube topology, but was unclear on larger topologies within the simulation environment. The advantages may not be as necessary for larger quadrants, as the number of minimal paths between source and destination increase as the quadrant sizes increase – therefore, for larger networks, Periphery Avoidance did not have a positive impact on CQR routing.

Further work should be done to analyze the complete impact of including these new enhancements for intermediate routing decisions while data progresses through a routable quadrant. This work has shown that enhancing CQR to include a function of periphery avoidance helped to avoid hotspots within the network.

7.1 Future Work

There are many possible recommendations for future work as it relates to this particular work, but some very interesting ideas include:

- *Analyzing the effects of layer-4 implementations.* Though this work used TCP exclusively within the laboratory tests, it would be worth analyzing the true impact of another layer-4 implementation.

- *Implementing a readily-available dynamic routing implementation such as Zebra and Quagga.* By analyzing the routing algorithms on another routing daemon, further validation could be achieved.

- *Implementing the dynamic algorithm by using Linux IP Routing Tables.* Though assumptions were made indicating that such an implementation would be computationally taxing, it would be worth analyzing further and seeing the direct impact of such an implementation.

- *Apply minimization or optimal control techniques.* By doing this, and applying weights towards the variables of periphery avoidance and output queues, an optimum decision function could be found to help further increase Enhanced CQR's ability to route around hotspots while effectively balancing the network load.

- *Analyzing other interconnect implementations such as ethernet, firewire, or SATA.* All had been discussed as possibilities, but the implementation of either could make a research cluster such as Thor's Tack Hammer more appealing to a variety of users.

- *Hybridizing this research with that of overlay networks.* By using these new fully-adaptive routing decisions, routing within overlay networks could benefit from this research.

- *Including the evaluation of flow control operations in the presence of hotspots and difficult traffic patterns.* Though this research direction was not included within this work, the implementation of good flow control mechanisms can definitely improve the efficiency of a parallel computing cluster. Further analysis combining the enhancements to CQR with good flow-control practices could further demonstrate the benefits of implementing adaptive routing algorithms.

- *Analyzing the effects of neighboring hotspots verses non-neighboring hotspots.* Though this was not analyzed within this work, analysis of hotspots of neighboring nodes could impact the results of any hotspot analysis.

- *Further analyze the scalability of Enhanced CQR with Periphery Avoidance.* As it was indicated within the previous chapter, larger networks did not indicate the benefits of using ECQR as they did with the 3-ary 3-cube network (both simulation and laboratory results). Also, including more emphasis in analytically determining at what percentage of hot-spots ECQR's benefits were clearly seen has yet to be found. This could also turn into another minimization or optimal control problem.

Bibliography

[1] N. Adiga, M. Blumrich, D. Chen, P. Coteus, and A. Gara. Blue gene/l torus interconnection network. *IBM J. Res. and Dev*, Jan 2005.

[2] G. Almasi, C. Archer, J. Castanos, and J. Gunnels. Design and implementation of message-passing services for the blue gene/l supercomputer. *IBM Journal of Research and Development*, Jan 2005.

[3] P. Berman, L. Gravano, G. Pifarre, and J. Sanz. Adaptive deadlock-and livelock-free routing with all minimal paths in torus networks. *Proceedings of the fourth annual ACM symposium on Parallel Algorithms and Architectures*, Jan 1992.

[4] M. Blumrich, D. Chen, P. Coteus, A. Gara, and M. Giampapa. Design and analysis of the bluegene/l torus interconnection network. *IBM Research Report RC23025 (W0312-022) December*, Jan 2003.

[5] K. Bolding, M. Fulgham, and L. Snyder. The case for chaotic adaptive routing. *Computers*, Jan 1997.

[6] Camp, B. Cole, E. DeBenedictis, R. Leland, J. Tomkins, and et. al. Architectural specification for massively parallel computers: an experience and measurement-based approach. *Concurrency and Computation Practice and Experience*, Jan 2005.

[7] W. Dally, H. Aoki, and C. MIT. Deadlock-free adaptive routing in multicomputer networks usingvirtual channels. *Parallel and Distributed Systems*, Jan 1993.

[8] W. J. Dally. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, Jan 2004.

[9] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. Morgan Kaufmann, Jan 2003.

[10] A. Hoisie, G. Johnson, D. Kerbyson, M. Lang, and S. Pakin. A performance comparison through benchmarking and modeling of three leading supercomputers: blue gene/l, red storm, and purple. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Jan 2006.

[11] D. Mills. A brief history of ntp time: confessions of an internet timekeeper. *ACM Computer Communications Review*, Jan 2003.

[12] A. Singh, W. Dally, A. Gupta, and B. Towles. Goal: a load-balanced adaptive routing algorithm for torus networks. *Computer Architecture*, Jan 2003.

[13] A. Singh, W. Dally, A. Gupta, and B. Towles. Adaptive channel queue routing on k-ary n-cubes. *Proceedings of the sixteenth annual ACM symposium on Parallel Algorithms and Architectures*, Jan 2004.

[14] A. Singh, W. Dally, B. Towles, and A. Gupta. Globally adaptive load-balanced routing on tori. *Computer Architecture Letters*, Jan 2004.

[15] L. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, Jan 1982.

[16] J. Vetter, S. Alam, T. D. Jr, and M. Fahey. Early evaluation of the cray xt3 at ornl. *Proceedings of the 47th Cray User Group Conference*, Jan 2005.

Appendix A

Appendix A: Linux BASH Scripts

A.1 USB Interconnect Scripts

This first section lists the BASH scripts used to correctly initialize the USB interconnection network on Thor's Tack Hammer. At the head of each script contains its functionality.

The file triplets.txt, which was used within these scripts but was not included, simply listed the hardware address for each node, and the corresponding x, y, and z values. It also included the order for which each interface was brought up within Linux. It was found that this was the same order for each node.

A.1.1 file: interconnectUp

```
#!/bin/bash

# +++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: interconnectUp
# Author: Chris Lydick
# Date: Feb 18, 2008
#
#  This script is run on the head-node and brings up
#   all nodes' usb interfaces and then pings them. All
#   information is routed to a log.txt file for analysis.
#
# +++++++++++++++++++++++++++++++++++++++++++++++++++++++ #


# These variables allow the script to color-code the success/failures.
NORMAL="^[[0;39m"
RED="^[[1;31m"
GREEN="^[[1;32m"

# Print the date at the top of the log.txt file.
echo `date` >> log.txt

# Copy the triplets.txt file if it is not in the default location.
if [ -e /pkhome/pkhome/triplets.txt ]; then
echo "/pkhome/pkhome/triplets.txt was found."
else
cp -f triplets.txt /pkhome/pkhome/triplets.txt
echo "/pkhome/pkhome/triplets.txt was not found."
echo "Copied successfully."
```

```
fi

# For all nodes in the LAM, bring up the interfaces. If
#   an '.oops.usbup' file was touched, we see that as a flag indicating an error.
for i in `cat /pkhome/pkhome/tmp/bhosts|head -n 27`
do
echo -n "Bringing up USB interfaces for $i..."
echo "Bringing up USB interfaces for $i..." >> log.txt
ssh knoppix@$i "sudo /pkhome/pkhome/usbUp" >> log.txt
if [ -e /pkhome/pkhome/tmp/.oops.usbup ]; then
echo "$RED failed. $NORMAL"
echo "failed." >> log.txt
rm -f /pkhome/pkhome/tmp/.oops.usbup
else
echo "$GREEN success. $NORMAL";
echo "success." >> log.txt
fi
done

# Wait 10 seconds for the routing tables to fully initialize before pinging
#  the neighbors.
sleep 10

# Now, ping the nodes... output all information to the logs.
for i in `cat /pkhome/pkhome/tmp/bhosts|head -n 27`
do
echo "Pinging hosts from $i."
echo "Pinging hosts from $i." >> log.txt
ssh knoppix@$i "sudo /pkhome/pkhome/usbPing" >> log.txt
done

# User interface.
echo ""
echo "You may want to verify all interfaces were brought up without"
echo "error by viewing the file log.txt."

# This makes partitioning the logs simple.
echo "==========================================================" >> log.txt
echo "" >> log.txt
echo "" >> log.txt
```

A.1.2 file: usbUp

```
#/bin/bash

# +++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: usbUp
# Author: Chris Lydick
# Date: Feb 18, 2008
#
#  This script brings up the USB Ethernet Interfaces and
#   automatically assigns the addresses/subnets.
#
# +++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

k=3
n=3
PRIMARYOCTET="10.0"
USBSUBNET="255.255.255.252"

getTriplet()
{

 # First find our management hardware address
 for i in ‘ifconfig eth2 |grep HWaddr‘
 do
if [ ‘echo $i |head -c 5‘ == "00:E0" ]; then
hwaddr=$i;
fi
 done

 # Given the hardware address, lookup our triplet information in a
 #   global file, "triplets.txt"
 triplet=‘cat /home/knoppix/triplets.txt |grep $hwaddr |tail -c 6‘

 # get x,y,z
 z=‘echo $triplet |head -c 1‘
 x=‘echo $triplet |tail -c 2‘
 y=‘echo $triplet |tail -c 4 |head -c 1‘
 echo "triplet: $x,$y,$z"

 # This next area allows for the triplets.txt file to contain
 #   information as to the specific order for usb0-5 as they’re
 #   brought up. If all nodes are correctly booted, they should
 #   have a consistent order.
 order=‘cat /home/knoppix/triplets.txt |grep $hwaddr |head -c 11 |tr ’.’ ’ ’‘
 first=‘echo ${order:0:1}‘;
 second=‘echo ${order:2:1}‘;
 third=‘echo ${order:4:1}‘;
 fourth=‘echo ${order:6:1}‘;
 fifth=‘echo ${order:8:1}‘;
 sixth=‘echo ${order:10:1}‘;
}

#  This function calculates both local and neighbor addresses
#     for the x-directions (+/-).
getXs()
{
 dir_mod_pos=1;
 dir_mod_neg=2;
 POSCOCTET=$[($x<<4)+$y];
 if [ $x -eq 0 ]; then
NEGCOCTET=$[(($k-1)<<4)+$y];
 else
NEGCOCTET=$[(($x-1)<<4)+$y];
 fi
```

```
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 NEGDOCTET=$[($z<<4)+$dir_mod_neg];
 POSXIP="$PRIMARYOCTET.$POSCOCTET.$POSDOCTET";
 NEGXIP="$PRIMARYOCTET.$NEGCOCTET.$NEGDOCTET";
 NEXTPOSX="$PRIMARYOCTET.$POSCOCTET.$[$POSDOCTET+1]";
 NEXTNEGX="$PRIMARYOCTET.$NEGCOCTET.$[$NEGDOCTET-1]";
 echo "The Positive X interface is: $POSXIP";
 echo "The Negative X interface is: $NEGXIP";
}

#  This function calculates both local and neighbor addresses
#     for the y-directions (+/-).
getYs()
{
 dir_mod_pos=5;
 dir_mod_neg=6;
 POSCOCTET=$[($x<<4)+$y];
 if [ $y -eq 0 ]; then
NEGCOCTET=$[($x<<4)+($k-1)];
 else
NEGCOCTET=$[($x<<4)+($y-1)];
 fi
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 NEGDOCTET=$[($z<<4)+$dir_mod_neg];
 POSYIP="$PRIMARYOCTET.$POSCOCTET.$POSDOCTET";
 NEGYIP="$PRIMARYOCTET.$NEGCOCTET.$NEGDOCTET";
 NEXTPOSY="$PRIMARYOCTET.$POSCOCTET.$[$POSDOCTET+1]";
 NEXTNEGY="$PRIMARYOCTET.$NEGCOCTET.$[$NEGDOCTET-1]";
 echo "The Positive Y interface is: $POSYIP";
 echo "The Negative Y interface is: $NEGYIP";
}

#  This function calculates both local and neighbor addresses
#     for the z-directions (+/-).
getZs()
{
 dir_mod_pos=9;
 dir_mod_neg=10;
 COCTET=$[($x<<4)+$y];
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 if [ $z -eq 0 ]; then
NEGDOCTET=$[(($k-1)<<4)+$dir_mod_neg];
 else
NEGDOCTET=$[(($z-1)<<4)+$dir_mod_neg];
 fi
 POSZIP="$PRIMARYOCTET.$COCTET.$POSDOCTET";
 NEGZIP="$PRIMARYOCTET.$COCTET.$NEGDOCTET";
 NEXTPOSZ="$PRIMARYOCTET.$COCTET.$[$POSDOCTET+1]";
 NEXTNEGZ="$PRIMARYOCTET.$COCTET.$[$NEGDOCTET-1]";
 echo "The Positive Z interface is: $POSZIP";
 echo "The Negative Z interface is; $NEGZIP";
}

getTriplet;
getXs;
getYs;
getZs;

#  List all usb interfaces in $usbs, and count
for i in ‘ifconfig -a |grep usb‘
do
if [ ‘echo $i |head -c 3‘ == "usb" ]; then
count=‘echo $[count+1]‘;
usbs="$usbs $i";
fi
```

```
done

#  Bring up usb interfaces
count_x=0;
if [ $count == 6 ]; then
#echo $usbs
for j in $usbs
do
echo -n "Bringing up interface $j...";
sudo ifconfig $j up;
case "$count_x" in
"$first" ) sudo ifconfig $j inet $POSYIP netmask $USBSUBNET;;
"$second" ) sudo ifconfig $j inet $NEGYIP netmask $USBSUBNET;;
"$third" ) sudo ifconfig $j inet $POSXIP netmask $USBSUBNET;;
"$fourth" ) sudo ifconfig $j inet $NEGXIP netmask $USBSUBNET;;
"$fifth" ) sudo ifconfig $j inet $NEGZIP netmask $USBSUBNET;;
"$sixth" ) sudo ifconfig $j inet $POSZIP netmask $USBSUBNET;;
* ) echo "Found an extra case in the loop.";;
esac
[ $? -eq 0 ] && echo "success." || touch /home/knoppix/tmp/.oops.usbup
count_x=`echo $[count_x+1]`;
done
#  This occurs when there aren't exactly 6 usb interfaces. Consider rebooting node.
#    A file is touched which can be seen by the head-node, indicating an error.
else
echo "A problem occurred. Not all interfaces appear to be working."
echo "Only $count interfaces were found."
echo "Exiting..."
touch /home/knoppix/tmp/.oops.usbup
fi
```

A.1.3 file: usbPing

```
#/bin/bash

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: usbPing
# Author: Chris Lydick
# Date: Feb 18, 2008
#
#  This script allows a node to ping all of its closest
#   neighbors (within 1 hop).
#
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

k=3
n=3

PRIMARYOCTET="10.0"
USBSUBNET="255.255.255.252"

getTriplet()
{
# First find our management hardware address
 for i in `ifconfig eth2 |grep HWaddr`
 do
if [ `echo $i |head -c 5` == "00:E0" ]; then
hwaddr=$i;
fi
 done

 # Given the hardware address, lookup our triplet information in a
 #   global file, "triplets.txt"
 triplet=`cat /home/knoppix/triplets.txt |grep $hwaddr |tail -c 6`

 # get x,y,z
 z=`echo $triplet |head -c 1`
 x=`echo $triplet |tail -c 2`
 y=`echo $triplet |tail -c 4 |head -c 1`
 echo "triplet: $x,$y,$z"
}

#  This function calculates both local and neighbor addresses
#     for the x-directions (+/-).
getXs()
{
 dir_mod_pos=1;
 dir_mod_neg=2;
 POSCOCTET=$[($x<<4)+$y];
 if [ $x -eq 0 ]; then
NEGCOCTET=$[(($k-1)<<4)+$y];
 else
NEGCOCTET=$[(($x-1)<<4)+$y];
 fi
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 NEGDOCTET=$[($z<<4)+$dir_mod_neg];
 POSXIP="$PRIMARYOCTET.$POSCOCTET.$POSDOCTET";
 NEGXIP="$PRIMARYOCTET.$NEGCOCTET.$NEGDOCTET";
 NEXTPOSX="$PRIMARYOCTET.$POSCOCTET.$[$POSDOCTET+1]";
 NEXTNEGX="$PRIMARYOCTET.$NEGCOCTET.$[$NEGDOCTET-1]";
}

#  This function calculates both local and neighbor addresses
#     for the y-directions (+/-).
getYs()
{
```

```
 dir_mod_pos=5;
 dir_mod_neg=6;
 POSCOCTET=$[($x<<4)+$y];
 if [ $y -eq 0 ]; then
NEGCOCTET=$[($x<<4)+($k-1)];
 else
NEGCOCTET=$[($x<<4)+($y-1)];
 fi
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 NEGDOCTET=$[($z<<4)+$dir_mod_neg];
 POSYIP="$PRIMARYOCTET.$POSCOCTET.$POSDOCTET";
 NEGYIP="$PRIMARYOCTET.$NEGCOCTET.$NEGDOCTET";
 NEXTPOSY="$PRIMARYOCTET.$POSCOCTET.$[$POSDOCTET+1]";
 NEXTNEGY="$PRIMARYOCTET.$NEGCOCTET.$[$NEGDOCTET-1]";
}

#  This function calculates both local and neighbor addresses
#     for the z-directions (+/-).
getZs()
{
 dir_mod_pos=9;
 dir_mod_neg=10;
 COCTET=$[($x<<4)+$y];
 POSDOCTET=$[($z<<4)+$dir_mod_pos];
 if [ $z -eq 0 ]; then
NEGDOCTET=$[(($k-1)<<4)+$dir_mod_neg];
 else
NEGDOCTET=$[(($z-1)<<4)+$dir_mod_neg];
 fi
 POSZIP="$PRIMARYOCTET.$COCTET.$POSDOCTET";
 NEGZIP="$PRIMARYOCTET.$COCTET.$NEGDOCTET";
 NEXTPOSZ="$PRIMARYOCTET.$COCTET.$[$POSDOCTET+1]";
 NEXTNEGZ="$PRIMARYOCTET.$COCTET.$[$NEGDOCTET-1]";
}

getTriplet;
getXs;
getYs;
getZs;

#  Ping all neighbors - we filter out all lines except where we successfully
#    or unsuccessfully transmitted one ICMP packet.
echo -n "+x $NEXTPOSX: "
ping -q -c 1 $NEXTPOSX |grep transmitted
echo -n "-x $NEXTNEGX: "
ping -q -c 1 $NEXTNEGX |grep transmitted
echo -n "+y $NEXTPOSY: "
ping -q -c 1 $NEXTPOSY |grep transmitted
echo -n "-y $NEXTNEGY: "
ping -q -c 1 $NEXTNEGY |grep transmitted
echo -n "+z $NEXTPOSZ: "
ping -q -c 1 $NEXTPOSZ |grep transmitted
echo -n "-z $NEXTNEGZ: "
ping -q -c 1 $NEXTNEGZ |grep transmitted
```

A.2 Data Analysis Scripts

The following scripts enabled efficient analysis of the data that was to be sent or had been sent across the USB Interconnect. As with before, the head of each script contains information describing its functionality.

A.2.1 file: interconnectTime

```
#!/bin/bash

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: interconnectTime
# Author: Chris Lydick
# Date: Mar 3, 2008
#
#  This script reports the estimated and maximum error
#   of time from the headnode (using NTP).
#
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++ #


for i in `cat /pkhome/pkhome/tmp/bhosts | head -n 27`
do
echo $i
ssh $i "ntptime | grep error"
done
```

A.2.2 file: interconnectRouters

```
#!/bin/bash

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: interconnectRouters
# Author: Chris Lydick
# Date: Mar 3, 2008
#
#  This script compiles the most recent version of server.c,
#   loserver.c, and injector.c, and executes the script nodeSetup
#   locally. It also grabs usb byte counts (line 27).
#
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

# This script first kills all loserver and server processes
#   and then restarts the server, sending output to /dev/null.
cd /mnt/sda2/usb/InterconnectScripts/

gcc -o /pkhome/pkhome/loserver server/loserver.c
gcc -o /pkhome/pkhome/server server/server.c -lm
gcc -o server/injector server/injector.c

j=26;
for i in ‘cat /pkhome/pkhome/tmp/bhosts |head -n 27‘
do
echo "$i, addr:$j"
ssh $i "/pkhome/pkhome/nodeSetup $j"
ssh $i "cat /proc/net/dev |grep usb" > server/logs/$i-before
j=$[$j-1];
done
```

A.2.3 file: nodeSetup

```
#!/bin/bash

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: nodeSetup
# Author: Chris Lydick
# Date: Mar 3, 2008
#
#  This script first kills all previous server/loserver
#    processes and then re-initializes them with the newly
#    compiled versions. It then sets the CPU affinity for
#    "server" exclusively to the first CPU and "loserver"
#    exclusively to the second CPU. All output is redirected
#    to /dev/null, and servers are placed in background.
#
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

sudo pkill server
/pkhome/pkhome/loserver $1 1>/dev/null &
/pkhome/pkhome/server $1 1>/dev/null &
taskset -p 01 ‘pgrep -x server‘ 1>/dev/null
taskset -p 02 ‘pgrep -x loserver‘ 1>/dev/null

exit

```

A.2.4 file: interconnectThroughput

```
#!/bin/bash

# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #
# File: interconnectThroughput
# Author: Chris Lydick
# Date: Mar 3, 2008
#
#  This script parses all data that was sent during a run
#   and returns statistics given their timestamps.
#
# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

#  This check verifies we've not run it since resetting the
#    USB byte counts.
if [ -e server/logs/192.168.0.200-before ]; then
echo -n ""
else
echo "No before files found. You must re-run interconnectRouters to regenerate these files."
exit;
fi

echo -n "Getting latest USB byte counts..."

#  For each node, get the latest USB byte counts.
for i in `cat /pkhome/pkhome/tmp/bhosts |head -n 27`
do
ssh $i "cat /proc/net/dev |grep usb" > server/logs/$i-after
done
echo "done."

allnums="0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26"
nums="1 2 3 4 5 6"
diffs=""

#  For each node...
for i in `cat /pkhome/pkhome/tmp/bhosts |head -n 27`
do
#  And for each of the 6 interfaces...
for j in $nums
do
#  Get the jth byte count at the start.
var1=`cat server/logs/$i-before |head -n $j |tail -n 1|tr ":" " "`
set -- $var1
shift; shift; shift; shift; shift; shift; shift; shift; shift;
trans_before=$1;

#  Get the jth byte count at the end of run.
var2=`cat server/logs/$i-after |head -n $j |tail -n 1|tr ":" " "`
set -- $var2
shift; shift; shift; shift; shift; shift; shift; shift; shift;
trans_after=$1;

#  Only analyze the transmitted data. To do both would be redundant.
trans=$[$trans_after-$trans_before];
diffs="$diffs $trans"
done

done

#  For ease, the differences between before-bytes and after-bytes can be copied/pasted
#    to Matlab.
echo "The following can be pasted to MatLab for a histogram of the link utilizations."
echo ""
```

```
echo ""
#diffs=`echo $diffs |tr " " "\n"`
echo "x = [$diffs]"
echo "x = x./(max(x))"
echo "x = [x 0]"
echo "hist(x,100)";
echo ""
echo ""

#  Remove these logs, force the user to re-run interconnectRouters script.
rm -fr server/logs/*

#  This loop was necessary for large numbers of packets. `cat` has a finite space
#   when it comes to data that is passed to it.
cd /pkhome/pkhome/router/routerlogs/
for i in $allnums
do
hop_1="$hop_1 `echo "" |cat \`grep -d recurse -l "hops: 1" $i/ \` |grep trans`"
hop_2="$hop_2 `echo "" |cat \`grep -d recurse -l "hops: 2" $i/ \` |grep trans`"
hop_3="$hop_3 `echo "" |cat \`grep -d recurse -l "hops: 3" $i/ \` |grep trans`"
avg_queue="$avg_queue `echo "" |cat \` find $i/ |grep SUCCESS\` |grep avg_queue`"
done

#  First find all data dropped by the C programs.
alldrops=`find . |grep DROP`
set -- $alldrops
echo "actual drops: $#";
rm -f `find . |grep DROP` || echo -n "";

#  Find all unknown data dropped by the C programs.
allunknown=`find . |grep UNKNOWN`
set -- $allunknown
echo "actual unknown: $#"
rm -f `find . |grep UNKNOWN` || echo -n "";

#  Parse the average queueing times.
set -- $avg_queue
total_drops=0;
queue=0;
total_packets=0;
i=0;
j=0;

while [ $# -gt 0 ]
do
shift
i=$[$i+1];
queue=$[$queue+$1];
shift
done
if [ $i -gt 0 ]; then
queue=$[$queue/$i];
echo "avg queue time: $queue"
fi

#  Parse the headers from data which traversed one hop.
set -- $hop_1
hops=0;
i=0;
j=0;
while [ $# -gt 0 ]
do
shift
# If the time was greater than 1.5 seconds, we
#   assume there was some kind of timeout. This
```

```
#   data will be discarded, counted as a drop.
if [ $1 -lt 1500000 ]; then
i=$[$i+1];
hops=$[$hops+$1];
else
j=$[$j+1];
fi
shift
done
total_packets=$[$total_packets+$j+$i];
if [ $i -gt 0 ]; then
hops=$[$hops/$i];
echo "avg 1-hop time: $hops"
fi
total_drops=$[$total_drops+$j];
all_avg=$[$i*$hops];

#  Parse the headers from data which traversed two hops.
set -- $hop_2
hops=0;
i=0;
j=0;
while [ $# -gt 0 ]
do
shift
if [ $1 -lt 1500000 ]; then
i=$[$i+1];
hops=$[$hops+$1];
else
j=$[$j+1];
fi
shift
done
total_packets=$[$total_packets+$j+$i];
if [ $i -gt 0 ]; then
hops=$[$hops/$i];
echo "avg 2-hop time: $hops"
fi
total_drops=$[$total_drops+$j];
all_avg=$[$all_avg + ($hops*$i)];

#  And finally, for three hops.
set -- $hop_3
hops=0;
i=0;
j=0;
while [ $# -gt 0 ]
do
shift
if [ $1 -lt 1500000 ]; then
i=$[$i+1];
hops=$[$hops+$1];
else
j=$[$j+1];
fi
shift
done
total_packets=$[$total_packets+$j+$i];
if [ $i -gt 0 ]; then
hops=$[$hops/$i];
echo "avg 3-hop time: $hops"
fi
total_drops=$[$total_drops+$j];
all_avg=$[($all_avg + ($hops*$i))/($total_packets-$total_drops)];
```

```
#  Echo all global information
echo "avg trans time: $all_avg";
echo "total drops: $total_drops"
echo "total packets: $[$total_packets/2]"

#  Remove all data packets, we're done.
cd /pkhome/pkhome/router/routerlogs/
rm -f `find . |grep SUCCESS`

```

Appendix B

Appendix B: C Programs

B.1 file: server.c

```
/* ############################################################ *
 * Filename: router.c     *
 * Author:   Chris Lydick     *
 * Date:     Mar 1, 2008 *
 * Usage:    ./server [address] *
 * Notes:    Portions borrowed from http://tinyurl.com/2w36o4   *
 * and http://tinyurl.com/6bu7s *
 *    This is a program which was created to route data over *
 *    a torus-mesh/hypercube network using: *
 *     1. Dimension Ordered Routing *
 *     2. Direction Ordered Routing *
 *     3. Minimal Adaptive Routing *
 *     4. Minimal Oblivious Routing *
 *     5. CQR Routing *
 *     6. Enhanced CQR *
 * ############################################################ */

#include "router.h"

/* ======================================== */
/* =========   FUNCT DECLARATION  ========= */
/* ======================================== */
void initNeighborhood(void);
void child_handler(int s);
void myperror(char *x);
int makeDecision(char *hdr, char *ra, char *dst);
void send_packet(char* packet, char* host, unsigned long long timestamp);
int dimord(int s, int d);
void sem_lock(int id);
void sem_unlock(int id);
int get_outputQueue(int sem_id, struct queue *q);
int inc_outputQueue(int sem_id, struct queue *q);
int dec_outputQueue(int sem_id, struct queue *q);
void adjust_queues(int sem_id, struct queue *q);
int min_choose(int x, int y, int z);
int adaptive_choose(int x, int y, int z);
int adaptive_periphery_choose(int x, int y, int z);
void make_vector(char v[], char dir[], int x_val, int y_val, int z_val);
int min(int x, int y);
double min_d(double x, double y);

/* ======================================== */
```

```
/* =========   GLOBAL VARIABLES   ========= */
/* ======================================= */
int myx, myy, myz, k, n;
char *addr, next_posx[12], next_posy[12], next_posz[12], next_negx[12], next_negy[12], next_negz[12];
struct queue *q_posx, *q_posy, *q_posz, *q_negx, *q_negy, *q_negz;
int sem_posx, sem_posy, sem_posz, sem_negx, sem_negy, sem_negz;

/* ======================================= */
/* =========    Main Function     ========= */
/* ======================================= */
// This is the main fuction which is executed.
// It's purpose is to:
// (1): Accept TCP Packets of fixed size on port 3490.
// (2): Analyzes the packet...
//  (a): if: Hop Count is higher than MAXHOP: drop, route to localhost.
//  (b): else if: packet is destined for this host: route to localhost
//  (c): else: make routing decision, and send packet onto next hop.
int main(int argc, char* argv[])
{
int sockfd, new_fd, numbytes, hop_nu, remSocket, yes;
int shmid;
struct sockaddr_in my_addr, their_addr, next_addr;
union semun sem1, sem2, sem3, sem4, sem5, sem6;
struct hostent *remHost;
struct timeval t;
struct sigaction sa;
char* shmem;
char newbuf[MAXDATASIZE], buf[MAXDATASIZE], header[HEADERSIZE+1];
char h_src[3], h_dest[3], h_ra[2], h_hops[3], h_queue[7];
char h_queue_t[7], d_time[7], time_ms_t[17], time_ms[10];
unsigned long long t1, t2;
socklen_t sin_size;


k=3; n=3; yes=1;
if (argc > 1)
addr = argv[1];
else myperror("usage: ./server [node number]");

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
myperror("socket");
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
perror("setsockopt");

// Create 6 semaphores, all with one semaphore each, readable by the owner.
//  these correspond to the semaphores for each output queue on each USB interface
sem_posx = semget(SID_POSX, 1, IPC_CREAT | 0600);
sem_posy = semget(SID_POSY, 1, IPC_CREAT | 0600);
sem_posz = semget(SID_POSZ, 1, IPC_CREAT | 0600);
sem_negx = semget(SID_NEGX, 1, IPC_CREAT | 0600);
sem_negy = semget(SID_NEGY, 1, IPC_CREAT | 0600);
sem_negz = semget(SID_NEGZ, 1, IPC_CREAT | 0600);

// Now, initialize the semaphores to 1.
sem1.val = 1;
if (semctl(sem_posx, 0, SETVAL, sem1) == -1)
myperror("sem1");
sem2.val = 1;
if (semctl(sem_posy, 0, SETVAL, sem2) == -1)
myperror("sem2");
sem3.val = 1;
if (semctl(sem_posz, 0, SETVAL, sem3) == -1)
myperror("sem3");
sem4.val = 1;
if (semctl(sem_negx, 0, SETVAL, sem4) == -1)
```

```
myperror("sem4");
sem5.val = 1;
if (semctl(sem_negy, 0, SETVAL, sem5) == -1)
myperror("sem5");
sem6.val = 1;
if (semctl(sem_negz, 0, SETVAL, sem6) == -1)
myperror("sem6");

// Create a shared memory block of appropriate size.
shmid = shmget(IPC_PRIVATE, sizeof(struct queue)*6, IPC_CREAT | IPC_EXCL | 0600);
// Grab the address where it exists.
shmem = shmat(shmid, NULL, 0);

// Adjust the queue pointers to the appropriate positions within
//  the shared memory.
q_posx = (struct queue*) shmem;
q_posy = (struct queue*) shmem+(sizeof(struct queue)*1);
q_posz = (struct queue*) shmem+(sizeof(struct queue)*2);
q_negx = (struct queue*) shmem+(sizeof(struct queue)*3);
q_negy = (struct queue*) shmem+(sizeof(struct queue)*4);
q_negz = (struct queue*) shmem+(sizeof(struct queue)*5);

// Set all the values of each queue to 0.
  q_posx[0].val = 0;
q_negx[0].val = 0;
q_posy[0].val = 0;
q_negy[0].val = 0;
q_posz[0].val = 0;
q_negz[0].val = 0;

// Grab the time, and set all lastUpdated to the value of time now.
gettimeofday(&t, NULL);
t1 = (unsigned long long)t.tv_sec * 1000000 + (unsigned long long)t.tv_usec;
q_posx[0].lastUpdated = t1;
q_posy[0].lastUpdated = t1;
q_posz[0].lastUpdated = t1;
q_negx[0].lastUpdated = t1;
q_negy[0].lastUpdated = t1;
q_negz[0].lastUpdated = t1;

// Setup my server listen port.
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(ROUTEPORT);
my_addr.sin_addr.s_addr = INADDR_ANY;
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

// Bind a socket to my listen port.
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr) == -1)
myperror("bind");
// Allow a backlog of connections to BACKLOG
if (listen(sockfd, BACKLOG) == -1)
myperror("listen");
// Enable child_handler to reap all dead children lingering.
sa.sa_handler = child_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1)
myperror("sigaction");
// Initialize our neighborhood - calculate all IP addresses of interfaces and neighbors
initNeighborhood();
// main while loop.
while(1) {
sin_size = sizeof their_addr;
// Accept a new connection, immediately fork, and let child do work.
if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1) {
```

```
perror("accept");
continue; }
if (!fork()) {
// child doesn’t need the listener socket.
close(sockfd);
// receive the packet from the sender.
if ((numbytes=recv(new_fd, buf, MAXDATASIZE-1, 0)) == -1)
perror("recv");
// get time the packet was received
gettimeofday(&t, NULL);
t1 = (unsigned long long)t.tv_sec * 1000000 + (unsigned long long)t.tv_usec;
memcpy(newbuf,buf,MAXDATASIZE);
strncpy(&header[0],&newbuf[0],HEADERSIZE); header[HEADERSIZE+1] = '\0';
strncpy(&h_dest[0],&newbuf[2],2); h_dest[2] = '\0';
strncpy(&h_ra[0],&newbuf[4],1); h_ra[1] = '\0';
strncpy(&h_hops[0],&newbuf[5],2); h_hops[2] = '\0';
strncpy(&h_queue[0],&newbuf[16],6); h_queue[6] = '\0';
strncpy(&time_ms[0],&newbuf[7],9); time_ms[9] = '\0';
// if the header does not contain a time value for beginning,
//  insert the timestamp now.
if(atoi(time_ms) == 0){
sprintf(time_ms_t, "%llu", t1);
strncpy(&time_ms[0],&time_ms_t[7],9); time_ms[9] = '\0';
strncpy(&newbuf[7],&time_ms[0],9);
}
// if this packet is destined for itself, or has exceeded the max
//  number of hops, route the header to localhost & remove from network.
if ((atoi(h_dest) == atoi(addr))||(atoi(h_hops) >= MAXHOPS)) {
remHost=gethostbyname("127.0.0.1");
remSocket=socket(AF_INET, SOCK_STREAM, 0);
next_addr.sin_family = AF_INET;
next_addr.sin_port = htons(LOPORT);
next_addr.sin_addr = *((struct in_addr *)remHost->h_addr);
gettimeofday(&t, NULL);
t2 = (unsigned long long)t.tv_sec * 1000000 + (unsigned long long)t.tv_usec;
if ((t2-t1) >= 10000)
sprintf(d_time, "%llu", t2-t1);
else
sprintf(d_time,"0%llu", t2-t1);
sprintf(h_queue_t, "%d", atoi(d_time) + atoi(h_queue));
strncpy (&header[16],&h_queue_t[0], 6);
if (connect(remSocket,(struct sockaddr *)&next_addr, sizeof(next_addr)) < 0)
perror("connecting to localhost");
if (send(remSocket,header,HEADERSIZE-1,0) < 0)
perror("sending to localhost");
close(remSocket);

}
else {
// Fork off children to run the adjust_queues in parallel. That way if one queue
//   is slow to return the semaphore, others aren’t blocked.
if (atoi(h_ra) > 4) {
if (!fork()){
if (!fork()){
if (!fork()){
if (!fork()){
if (!fork()){
if (!fork()){
adjust_queues(sem_posx, q_posx);
exit(0);}
adjust_queues(sem_negx, q_negx);
exit(0);}
adjust_queues(sem_posy, q_posy);
exit(0);}
adjust_queues(sem_negy, q_negy);
```

```
exit(0);}
adjust_queues(sem_posz, q_posz);
exit(0);}
adjust_queues(sem_negz, q_negz);
exit(0);}
}
// add one to the hop.
newbuf[6] = newbuf[6]+1;
switch(makeDecision(newbuf,h_ra,h_dest)){
case GO_POSX:
// send the packet to the next hop in the +x direction
send_packet(newbuf, next_posx, t1);
// If a dynamic algorithm is being used, increment queue.
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_posx, q_posx);
printf("incrementing posx\n");}
break;
case GO_NEGX:
send_packet(newbuf, next_negx, t1);
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_negx, q_negx);
printf("incrementing negx\n");}
break;
case GO_POSY:
send_packet(newbuf, next_posy, t1);
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_posy, q_posy);
printf("incrementing posy\n");}
break;
case GO_NEGY:
send_packet(newbuf, next_negy, t1);
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_negy, q_negy);
printf("incrementing negy\n");}
break;
case GO_POSZ:
send_packet(newbuf, next_posz, t1);
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_posz, q_posz);
printf("incrementing posz\n");}
break;
case GO_NEGZ:
send_packet(newbuf, next_negz, t1);
if (atoi(h_ra) > 3) {
inc_outputQueue(sem_negz, q_negz);
printf("incrementing negz\n");}
break;
default:
printf("Chose the default area...\n");
remHost=gethostbyname("127.0.0.1");
remSocket=socket(AF_INET, SOCK_STREAM, 0);
next_addr.sin_family = AF_INET;
next_addr.sin_port = htons(LOPORT);
next_addr.sin_addr = *((struct in_addr *)remHost->h_addr);
gettimeofday(&t, NULL);
t2 = (unsigned long long)t.tv_sec * 1000000 + (unsigned long long)t.tv_usec;
if ((t2-t1) >= 10000)
sprintf(d_time, "%llu", t2-t1);
else
sprintf(d_time,"0%llu", t2-t1);
sprintf(h_queue_t, "%d", atoi(d_time) + atoi(h_queue));
strncpy (&header[16],&h_queue_t[0], 6);
if (connect(remSocket,(struct sockaddr *)&next_addr, sizeof(next_addr)) < 0)
perror("connecting to localhost");
if (send(remSocket,header,HEADERSIZE-1,0) < 0)
```

```
perror("sending to localhost");
close(remSocket);
break;


}

}
// Child is now done, close socket and exit.
close(new_fd);
exit(0);
}
// Parent doesn't need this.
close(new_fd);
}
return 0;
}

/* ====================================== */
/* ======== funct: makeDecision() ========= */
/* ====================================== */
// This function given the header, routing algorithm
//  and the destination, calculates the next hop.
//  It also takes into account any output queues if
//  a dynamic algorithm is used.
int makeDecision(char *hdr, char *ra, char *dest)
{
int x,y,z;
int db_addr;
int dir_x, dir_y, dir_z, algorithm;
int ret_val = -1;
char h_dir[4], h_v[4], h_int[4], h_x[2], h_y[2], h_z[2];
int sign_x, sign_y, sign_z;

algorithm = atoi(ra);
db_addr = (int)atoi(dest);
z = (int) floor(db_addr / (k*k));
y = (int) floor(db_addr / k) % k;
x = (int) db_addr % k;

switch (algorithm) {

case 0:  // Not Used.
break;
case 1:  // Dimension Ordered Routing
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
if ((dir_x=dimord(myx,x))!=0)
if (dir_x == -1)
ret_val = GO_NEGX;
else
ret_val = GO_POSX;
else if ((dir_y=dimord(myy,y))!=0)
if (dir_y == -1)
ret_val = GO_NEGY;
else
ret_val = GO_POSY;
else if ((dir_z=dimord(myz,z))!=0)
if (dir_z == -1)
ret_val = GO_NEGZ;
else
ret_val = GO_POSZ;

break;
```

```
case 2:  // Direction Ordered Routing
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
if (dir_x == 1)
ret_val = GO_POSX;
else if (dir_y == 1)
ret_val = GO_POSY;
else if (dir_z == 1)
ret_val = GO_POSZ;
else if (dir_x == -1)
ret_val = GO_NEGX;
else if (dir_y == -1)
ret_val = GO_NEGY;
else if (dir_z == -1)
ret_val = GO_NEGZ;

break;
case 3:  // Minimal Oblivious
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
ret_val = min_choose(dir_x, dir_y, dir_z);
break;
case 4:  // Minimal Adaptive
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
ret_val = adaptive_choose(dir_x, dir_y, dir_z);
break;
case 5:  // CQR
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
strncpy(&h_v[0],&hdr[22],3); h_v[3]='\0';
strncpy(&h_dir[0],&hdr[25],3); h_dir[3]='\0';
if (atoi(h_v) == 0) { // First hop, find v.
make_vector(h_v, h_dir, dir_x, dir_y, dir_z);
strncpy(&hdr[25],&h_dir[0],3);
strncpy(&hdr[22],&h_v[0],3);
}
h_x[0] = h_v[0]; h_x[1] = '\0';
h_y[0] = h_v[1]; h_y[1] = '\0';
h_z[0] = h_v[2]; h_z[1] = '\0';
if (h_dir[0] == '0') sign_x = 1;
else sign_x = -1;
if (h_dir[1] == '0') sign_y = 1;
else sign_y = -1;
if (h_dir[2] == '0') sign_z = 1;
else sign_z = -1;
ret_val = adaptive_choose(sign_x*atoi(h_x), sign_y*atoi(h_y), sign_z*atoi(h_z));
switch (ret_val) {
case GO_POSX: h_v[0] = h_v[0] - 1; break;
case GO_NEGX: h_v[0] = h_v[0] - 1; break;
case GO_POSY: h_v[1] = h_v[1] - 1; break;
case GO_NEGY: h_v[1] = h_v[1] - 1; break;
case GO_POSZ: h_v[2] = h_v[2] - 1; break;
case GO_NEGZ: h_v[2] = h_v[2] - 1; break;
}
strncpy(&hdr[22],&h_v[0],3);
break;
case 6: // CQR - Periphery Avoidance
dir_x = dimord(myx,x);
dir_y = dimord(myy,y);
dir_z = dimord(myz,z);
```

```
strncpy(&h_v[0],&hdr[22],3); h_v[3]='\0';
strncpy(&h_dir[0],&hdr[25],3); h_dir[3]='\0';
if (atoi(h_v) == 0) { // First hop, find v.
make_vector(h_v, h_dir, dir_x, dir_y, dir_z);
strncpy(&hdr[25],&h_dir[0],3);
strncpy(&hdr[22],&h_v[0],3);
}
h_x[0] = h_v[0]; h_x[1] = '\0';
h_y[0] = h_v[1]; h_y[1] = '\0';
h_z[0] = h_v[2]; h_z[1] = '\0';
if (h_dir[0] == '0') sign_x = 1;
else sign_x = -1;
if (h_dir[1] == '0') sign_y = 1;
else sign_y = -1;
if (h_dir[2] == '0') sign_z = 1;
else sign_z = -1;
ret_val = adaptive_periphery_choose(sign_x*atoi(h_x), sign_y*atoi(h_y), sign_z*atoi(h_z));
switch (ret_val) {
case GO_POSX: h_v[0] = h_v[0] - 1; break;
case GO_NEGX: h_v[0] = h_v[0] - 1; break;
case GO_POSY: h_v[1] = h_v[1] - 1; break;
case GO_NEGY: h_v[1] = h_v[1] - 1; break;
case GO_POSZ: h_v[2] = h_v[2] - 1; break;
case GO_NEGZ: h_v[2] = h_v[2] - 1; break;
}
strncpy(&hdr[22],&h_v[0],3);
break;
case 7: // VGD-CQR
break;
default:
break;
}
return ret_val;
}

/* ======================================= */
/* ======= funct: initNeighborhood() ====== */
/* ======================================= */
// This function initializes the IP addresses of
//  all the USB interfaces on this node, as well as
//  calculating the IP addresses of each neighbor.
void initNeighborhood(void)
{

char c_octet_pos[3], c_octet_neg[3];
char d_octet_pos[3], d_octet_neg[3];
int x,y,z;
double db_addr, res;
int dir_mod_x = 1;
int dir_mod_y = 5;
int dir_mod_z = 9;

db_addr = (double)atoi(addr);
z = (int) floor(db_addr / (k*k));
y = (int) floor(db_addr / k) % k;
x = (int) db_addr % k;
myx = x;
myy = y;
myz = z;

//Xs.
sprintf(c_octet_pos,"%d",((x*16)+y));
sprintf(d_octet_pos,"%d",((z*16)+dir_mod_x)+1);
sprintf(d_octet_neg,"%d",((z*16)+dir_mod_x));
if (!x)
```

```
sprintf(c_octet_neg,"%d",(((k-1)*16)+y));
else
sprintf(c_octet_neg,"%d",(((x-1)*16)+y));
sprintf(next_posx,"10.0.%s.%s",c_octet_pos,d_octet_pos);
sprintf(next_negx,"10.0.%s.%s",c_octet_neg,d_octet_neg);

//Ys.
sprintf(c_octet_pos,"%d",((x*16)+y));
sprintf(d_octet_pos,"%d",((z*16)+dir_mod_y)+1);
sprintf(d_octet_neg,"%d",((z*16)+dir_mod_y));
if (!y)
sprintf(c_octet_neg,"%d",((x*16)+(k-1)));
else
sprintf(c_octet_neg,"%d",((x*16)+(y-1)));
sprintf(next_posy,"10.0.%s.%s",c_octet_pos,d_octet_pos);
sprintf(next_negy,"10.0.%s.%s",c_octet_neg,d_octet_neg);

//Zs.
sprintf(c_octet_pos,"%d",((x*16)+y));
sprintf(c_octet_neg,"%d",((x*16)+y));
sprintf(d_octet_pos,"%d",((z*16)+dir_mod_z)+1);
if (!z)
sprintf(d_octet_neg,"%d",((k-1)*16)+dir_mod_z);
else
sprintf(d_octet_neg,"%d",((z-1)*16)+dir_mod_z);
sprintf(next_posz,"10.0.%s.%s",c_octet_pos,d_octet_pos);
sprintf(next_negz,"10.0.%s.%s",c_octet_neg,d_octet_neg);
}

/* ====================================== */
/* ========= funct: send_packet() ========= */
/* ====================================== */
// This function sends a packet to the correct destination.
//  Much of this code was originally repeated throughout this
//  file, all converged here.
void send_packet(char* packet, char* host, unsigned long long timestamp)
{
int remSocket;
struct timeval t;
struct hostent *remHost;
struct sockaddr_in next_addr;
unsigned long long t2;
char d_time[7], h_queue_t[7], h_queue[7];

// copy the previous queue value from the packet
strncpy(&h_queue[0],&packet[16],6); h_queue[6] = '\0';
remHost=gethostbyname(host);
// setup the outgoing socket
remSocket=socket(AF_INET, SOCK_STREAM, 0);
next_addr.sin_family = AF_INET;
next_addr.sin_port = htons(ROUTEPORT);
next_addr.sin_addr = *((struct in_addr *)remHost->h_addr);
// connect to the next hop, TCP handshake occurs
if(connect(remSocket,(struct sockaddr *)&next_addr,
sizeof(next_addr)) < 0)
myperror("connecting to next host");
// grab time & calculate difference, add to previous queue times
gettimeofday(&t, NULL);
t2 = ((unsigned long long)t.tv_sec * 1000000 +
(unsigned long long)t.tv_usec) - timestamp;
if (t2 >= 10000)
sprintf(d_time, "%llu", t2);
else
sprintf(d_time,"0%llu", t2);
sprintf(h_queue_t, "%d", atoi(d_time) + atoi(h_queue));
```

```
strncpy (&packet[16],&h_queue_t[0], 6);
// send data, close socket.
if(send(remSocket,packet,MAXDATASIZE-1,0) < 0)
myperror("sending to next host");
close(remSocket);
}

/* ======================================= */
/* =======  funct: child_handler() ======== */
/* ======================================= */
// This function reaps all dead child processes.
void child_handler(int s)
{
while(waitpid(-1, NULL, WNOHANG) > 0);
}

/* ======================================= */
/* =========  funct: myperror() =========== */
/* ======================================= */
// This function was created because these two
//  lines were used frequently.
void myperror(char *x)
{
perror(x);
exit(1);
}

/* ======================================= */
/* ==========  funct: dimord() ============ */
/* ======================================= */
// This function returns the difference between
//  si and di in a particular dimension.
int dimord(int s, int d)
{
int temp;
int ret_val;

temp = (d-s) % k;
if (temp < -1)
temp = temp + k;
else if (temp > 1)
temp = temp - k;

if (temp < 0)
ret_val = -1;
else if (temp > 0)
ret_val = 1;
if (s == d)
ret_val = 0;

return ret_val;
}

/* ======================================= */
/* ======  funct: get_outputQueue() ======= */
/* ======================================= */
// This function returns the value of the output
//  queue after obtaining the semaphore.
int get_outputQueue(int sem_id, struct queue *q)
{
int ret_val = -1;
// Receive the semaphore
sem_lock(sem_id);
// Enter the critical section
ret_val = q[0].val;
```

```
printf("queue val: %d\n", ret_val);
// Return the semaphore
sem_unlock(sem_id);

return ret_val;
}

/* ======================================= */
/* ======  funct: inc_outputQueue() ======= */
/* ======================================= */
// This function increments an output queue once
//  it successfully receives the semaphore for that
//  queue. It returns -1 if the queue is full.
int inc_outputQueue(int sem_id, struct queue *q)
{
int ret_val = -1;
// Receive the semaphore
sem_lock(sem_id);
// Enter the critical section
if (q[0].val < MAX_QUEUE)
{
q[0].val = q[0].val + 1;
ret_val = 1;
}
else printf("queue exceeded MAX\n");
// Return the semaphore
sem_unlock(sem_id);

return ret_val;
}

/* ======================================= */
/* =========funct: adjust_queues()========= */
/* ======================================= */
// This function adjusts the output queues by decrementing
//  the queues based on their last updated time value. If
//  it is greater than ADJUSTMENT, it is decreased that
//  number of times. It is assumed that packets depart
//  from the queues at a rate of one per ADJUSTMENT microseconds.
void adjust_queues(int sem_id, struct queue *q)
{
struct timeval t;
unsigned long long time;

gettimeofday(&t, NULL);
// Receive the semaphore
sem_lock(sem_id);
// Enter the critical section
time = (unsigned long long)t.tv_sec * 1000000 + (unsigned long long)t.tv_usec;
// remove as many packets from the output queues as we're expecting packets to leave.
while ((q[0].lastUpdated + ((unsigned long long)ADJUSTMENT)) <= time)
{
if (q[0].val > 0) q[0].val = q[0].val - 1;
q[0].lastUpdated = q[0].lastUpdated + ((unsigned long long)ADJUSTMENT);
}
// Return semaphore
sem_unlock(sem_id);

}

/* ======================================= */
/* =========  funct: sem_lock()   ========= */
/* ======================================= */
// This function locks a given semaphore. It blocks
//  until successfully obtained.
```

```
void sem_lock(int sem_set_id)
{
    struct sembuf sem_op;
    sem_op.sem_num = 0;
    sem_op.sem_op = -1;
    sem_op.sem_flg = 0;
    semop(sem_set_id, &sem_op, 1);
}

/* ======================================= */
/* =========  funct: sem_unlock() ========= */
/* ======================================= */
// This function returns a semaphore for use by
//  another process.
void sem_unlock(int sem_set_id)
{
    struct sembuf sem_op;
    sem_op.sem_num = 0;
    sem_op.sem_op = 1;
    sem_op.sem_flg = 0;
    semop(sem_set_id, &sem_op, 1);
}

/* ======================================= */
/* =========  funct: min_choose() ========= */
/* ======================================= */
// This function randomly chooses an order based on
//  all permutations of k and then returns which
//  direction to randomly route within given the
//  possible values passed. Eg. if a packet can
//  minimally route +x or -y, this function picks
//  between the two choices.
int min_choose(int x, int y, int z)
{
int t1;
int ret_val = 0;

if ((t1=rand())<0.166)
{
if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (z != 0)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;
}
else if (t1 < 0.333)
{

if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (z != 0)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;
else if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
}
else if (t1 < 0.5 )
{
if (z != 0)
```

```
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;
else if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
}
else if (t1 < 0.666)
{
if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (z != 0)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;
else if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
}
else if (t1 < 0.866)
{

if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (z != 0)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;

}
else
{
if (z != 0)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;
else if (y != 0)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (x != 0)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;

}
return ret_val;

}



/* ======================================== */
/* ======= funct: adaptive_choose() ======= */
/* ======================================== */
// This is the function which calculates the next
//  adaptive decision based on the possible directions
//  and the output queues.
int adaptive_choose(int x, int y, int z)
{
int t, ret_val;
int x_queue = LARGENU;
int y_queue = LARGENU;
```

```
int z_queue = LARGENU;

ret_val = -1;
if (x != 0)
if (x > 0)
x_queue = get_outputQueue(sem_posx, q_posx);
else
x_queue = get_outputQueue(sem_negx, q_negx);
if (y != 0)
if (y > 0)
y_queue = get_outputQueue(sem_posy, q_posy);
else
y_queue = get_outputQueue(sem_negy, q_negy);
if (z != 0)
if (z > 0)
z_queue = get_outputQueue(sem_posz, q_posz);
else
z_queue = get_outputQueue(sem_negz, q_negz);


t = min(min(x_queue, y_queue), z_queue);

if (t == x_queue)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (t == y_queue)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (t == z_queue)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;

return ret_val;
}

/* ================================================= */
/* ====== funct: adaptive_periphery_choose() ======= */
/* ================================================= */
// This function does the same as adaptive_choose, but
//  includes the periphery calculation in the determination
//  of the next hop.
int adaptive_periphery_choose(int x, int y, int z)
{

int ret_val;
double t, delta_total;
double x_queue = (double)LARGENU;
double y_queue = (double)LARGENU;
double z_queue = (double)LARGENU;


delta_total = (double)(abs(x) + abs(y) + abs(z));
ret_val = -1;
if (x != 0)
if (x > 0)
x_queue = (double)(1 + get_outputQueue(sem_posx, q_posx));
else
x_queue = (double)(1 + get_outputQueue(sem_negx, q_negx));
if (y != 0)
if (y > 0)
y_queue = (double)(1 + get_outputQueue(sem_posy, q_posy));
else
y_queue = (double)(1 + get_outputQueue(sem_negy, q_negy));
if (z != 0)
if (z > 0)
```

```
z_queue = (double)(1 + get_outputQueue(sem_posz, q_posz));
else
z_queue = (double)(1 + get_outputQueue(sem_negz, q_negz));

x_queue = x_queue * (1.0 - ((double)abs(x)/delta_total));
y_queue = y_queue * (1.0 - ((double)abs(y)/delta_total));
z_queue = z_queue * (1.0 - ((double)abs(z)/delta_total));

t = min_d(min_d(x_queue, y_queue), z_queue);

if (t == x_queue)
if (x > 0) ret_val = GO_POSX;
else ret_val = GO_NEGX;
else if (t == y_queue)
if (y > 0) ret_val = GO_POSY;
else ret_val = GO_NEGY;
else if (t == z_queue)
if (z > 0) ret_val = GO_POSZ;
else ret_val = GO_NEGZ;

return ret_val;

}


/* ======================================== */
/* ========= funct: make_vector() ========= */
/* ======================================== */
// This function creates the v and dir vectors for CQR and ECQR.
void make_vector(char v[], char dir[], int x_val, int y_val, int z_val){

int t_q, x_val_t, y_val_t, z_val_t;
double temp_min_val, min_val;
char v_1[4];
int nu_queues = 3;

// for +++
x_val_t = x_val;
y_val_t = y_val;
z_val_t = z_val;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
min_val=((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
sprintf(dir, "000");

// for ++-
x_val_t = x_val;
```

```
y_val_t = y_val;
if (z_val > 0) z_val_t = z_val-k;
else if (z_val < 0) z_val_t = z_val+k;
else z_val_t = 0;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}

// +-+
x_val_t = x_val;
if (y_val > 0) y_val_t = y_val-k;
else if (y_val < 0) y_val_t = y_val+k;
else y_val_t = 0;
z_val_t = z_val;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
```

```
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}


// +--
x_val_t = x_val;
if (y_val > 0) y_val_t = y_val-k;
else if (y_val < 0) y_val_t = y_val+k;
else y_val_t = 0;
if (z_val > 0) z_val_t = z_val-k;
else if (z_val < 0) z_val_t = z_val+k;
else z_val_t = 0;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}


// -++
if (x_val > 0) x_val_t = x_val-k;
else if (x_val < 0) x_val_t = x_val+k;
else x_val_t = 0;
y_val_t = y_val;
z_val_t = z_val;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
```

```
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}


// -+-
if (x_val > 0) x_val_t = x_val-k;
else if (x_val < 0) x_val_t = x_val+k;
else x_val_t = 0;
y_val_t = y_val;
if (z_val > 0) z_val_t = z_val-k;
else if (z_val < 0) z_val_t = z_val+k;
else z_val_t = 0;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}


//--+
if (x_val > 0) x_val_t = x_val-k;
else if (x_val < 0) x_val_t = x_val+k;
else x_val_t = 0;
if (y_val > 0) y_val_t = y_val-k;
else if (y_val < 0) y_val_t = y_val+k;
else y_val_t = 0;
z_val_t = z_val;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
```

```
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}


// ---
if (x_val > 0) x_val_t = x_val-k;
else if (x_val < 0) x_val_t = x_val+k;
else x_val_t = 0;
if (y_val > 0) y_val_t = y_val-k;
else if (y_val < 0) y_val_t = y_val+k;
else y_val_t = 0;
if (z_val > 0) z_val_t = z_val-k;
else if (z_val < 0) z_val_t = z_val+k;
else z_val_t = 0;
nu_queues = 3;
t_q = 0;
if (x_val_t > 0)
t_q = get_outputQueue(sem_posx, q_posx);
else if (x_val_t < 0)
t_q = get_outputQueue(sem_negx, q_negx);
else
nu_queues = nu_queues - 1;
if (y_val_t > 0)
t_q = t_q + get_outputQueue(sem_posy, q_posy);
else if (y_val_t < 0)
t_q = t_q + get_outputQueue(sem_negy, q_negy);
else
nu_queues = nu_queues - 1;
if (z_val_t > 0)
t_q = t_q + get_outputQueue(sem_posz, q_posz);
else if (z_val_t < 0)
t_q = t_q + get_outputQueue(sem_negz, q_negz);
else
nu_queues = nu_queues - 1;
temp_min_val = ((double)abs(x_val_t)+(double)abs(y_val_t)+(double)abs(z_val_t))*((double)t_q/(double)nu_queues);
if (temp_min_val < min_val){
min_val = temp_min_val;
sprintf(v_1, "%d%d%d", abs(x_val_t), abs(y_val_t), abs(z_val_t));
if (x_val_t >= 0) dir[0] = '0'; else dir[0] = '1';
if (y_val_t >= 0) dir[1] = '0'; else dir[1] = '1';
if (z_val_t >= 0) dir[2] = '0'; else dir[2] = '1';
}
v[0] = v_1[0];
v[1] = v_1[1];
v[2] = v_1[2];
```

```
}

/* ======================================== */
/* ========     funct: min_d()    ========= */
/* ======================================== */
// This function returns the minimum of the two values passed
double min_d(double x, double y)
{
return (double)((x*(x<y)) + (y*(x>=y)));
}

/* ======================================== */
/* ========      funct: min()     ========= */
/* ======================================== */
// This function returns the minimum of the two values passed
int min(int x, int y)
{
return ((x*(x<y)) + (y*(x>=y)));
}
```

B.2 file: loserver.c

```
/* ############################################################ *
 * Filename: loserver.c     *
 * Author:   Chris Lydick     *
 * Date:     Mar 1, 2008 *
 * Usage:    ./loserver [address] *
 * Notes:    Portions borrowed from http://tinyurl.com/2w36o4   *
 * *
 * ############################################################ */

#include "router.h"

/* ======================================== */
/* =========   FUNCT DECLARATION  ========= */
/* ======================================== */
void child_handler(int s);
void myperror(char *x);


/* ======================================== */
/* =========    Main Function     ========= */
/* ======================================== */
int main(int argc, char* argv[])
{
int sockfd, new_fd, numbytes, hop_nu, remSocket, yes;
struct sockaddr_in my_addr, their_addr, next_addr;
struct hostent *localhost;
socklen_t sin_size;
struct sigaction sa;
char newheader[HEADERSIZE+1], header[HEADERSIZE+1],  h_src[3], h_dest[3], h_ra[2], h_hops[3], *addr, fullpath[80];
char time_ms[10], time_ms_t[17], h_queue[7];
char h_time[10];
FILE *fp;
struct timeval tv;
time_t curtime;
int p_time, f_time, delta;
unsigned long long time;

yes=1;
if (argc > 1)
addr = argv[1];
else myperror("usage: ./loserver [node number]");

// Create server listen socket
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
myperror("socket");
// Set option on socket to reuse the address.
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
perror("setsockopt");

// Set parameters for binding the socket to an address.
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(LOPORT);
my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

// Bind the socket and address.
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr) == -1)
myperror("bind");

// Set the socket to listen for incoming connections.
if (listen(sockfd, BACKLOG) == -1)
myperror("listen");
```

```
// This reaps all dead child processes.
sa.sa_handler = child_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1)
myperror("sigaction");

// Main accept() loop
while(1) {
sin_size = sizeof their_addr;
if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1) {
perror("accept");
continue; }
printf("server: got connection from %s\n",inet_ntoa(their_addr.sin_addr));
// Fork off a child to do the work, only child will continue here.
if (!fork()) {
// Child process doesn't need the listener
close(sockfd);
if ((numbytes=recv(new_fd, header, HEADERSIZE-1, 0)) == -1)
perror("recv");
close(new_fd);
memcpy(newheader, header, HEADERSIZE); newheader[HEADERSIZE-1] = '\0';
strncpy(&h_src[0], &newheader[0],2); h_src[2] = '\0';
strncpy(&h_dest[0],&newheader[2],2); h_dest[2] = '\0';
strncpy(&h_ra[0],&newheader[4],1); h_ra[1] = '\0';
strncpy(&h_hops[0],&newheader[5],2); h_hops[2] = '\0';
strncpy(&h_time[0],&newheader[7],9); h_time[9] = '\0';
strncpy(&h_queue[0],&newheader[16],6); h_queue[6] = '\0';
gettimeofday(&tv, NULL);
time = (unsigned long long)tv.tv_sec * 1000000 + (unsigned long long)tv.tv_usec;
sprintf(time_ms_t,"%llu", time);
//sprintf(time_ms_t, "%ld.%ld", tv.tv_sec, tv.tv_usec);
strncpy(&time_ms[0],&time_ms_t[7],9); time_ms[9] = '\0';
//printf("time: %s\n", time_ms);
f_time = atoi(time_ms);
p_time = atoi(h_time);
delta = f_time - p_time;
//printf("f_time: %4.4f\n", f_time);
//printf("p_time: %4.4f\n", p_time);
//printf("time difference: %4.4f\n", f_time-p_time);
if (atoi(h_dest) == atoi(addr)){
sprintf(fullpath, "%s%s/SUCCESS_%s",FILE_PATH,addr,time_ms);
fp = fopen(fullpath, "w");
fprintf(fp,"from: %d\n", atoi(h_src));
fprintf(fp,"to: %d\n", atoi(h_dest));
fprintf(fp,"algorithm: %d\n", atoi(h_ra));
fprintf(fp,"hops: %d\n",atoi(h_hops));
fprintf(fp,"total_time: %d\n", delta);
fprintf(fp,"queue_time: %d\n", atoi(h_queue));
fprintf(fp,"trans_time: %d\n", delta-atoi(h_queue));
fprintf(fp,"avg_trans/hop: %d\n", (delta-atoi(h_queue))/atoi(h_hops));
fprintf(fp,"avg_queue/hop: %d\n", atoi(h_queue)/atoi(h_hops));
fprintf(fp,"start: %d\n", p_time);
fprintf(fp,"end: %d\n", f_time);
fprintf(fp,"header: %s\n", newheader);
fclose(fp);
}
else if (atoi(h_hops) >= MAXHOPS){
sprintf(fullpath, "%s%s/DROP_%s",FILE_PATH,addr,time_ms);
fp = fopen(fullpath, "w");
fprintf(fp,"from: %d\n", atoi(h_src));
fprintf(fp,"to: %d\n", atoi(h_dest));
fprintf(fp,"algorithm: %d\n", atoi(h_ra));
fprintf(fp,"hops: %d\n",atoi(h_hops));
```

```
fprintf(fp,"total_time: %d\n", delta);
fprintf(fp,"queue_time: %d\n", atoi(h_queue));
fprintf(fp,"trans_time: %d\n", delta-atoi(h_queue));
fprintf(fp,"start: %d\n", p_time);
fprintf(fp,"end: %d\n", f_time);
fprintf(fp,"header: %s\n", newheader);
fclose(fp);

}
else {
//perror("unrecognized packet");
sprintf(fullpath, "%s%s/UNKNOWN_%s",FILE_PATH,addr,time_ms);
fp = fopen(fullpath, "w");
fprintf(fp,"from: %d\n", atoi(h_src));
fprintf(fp,"to: %d\n", atoi(h_dest));
fprintf(fp,"algorithm: %d\n", atoi(h_ra));
fprintf(fp,"hops: %d\n",atoi(h_hops));
fprintf(fp,"total_time: %d\n", delta);
fprintf(fp,"queue_time: %d\n", atoi(h_queue));
fprintf(fp,"trans_time: %d\n", delta-atoi(h_queue));
fprintf(fp,"start: %d\n", p_time);
fprintf(fp,"end: %d\n", f_time);
fprintf(fp,"header: %s\n", newheader);
fclose(fp);
}
//printf("Closing this connection.");
exit(0);
}
close(new_fd);  // parent doesn't need this
}

return 0;
}


void child_handler(int s)
{
while(waitpid(-1, NULL, WNOHANG) > 0);
}

void myperror(char *x)
{
perror(x);
exit(1);
}
```

B.3 file: injector.c

```
/* ###########################################################  *
 * Filename: injector.c      *
 * Author:   Chris Lydick      *
 * Date:     Mar 1, 2008  *
 * Usage:    ./injector [traffic demand file] [routing algorithm]*
 * Notes:    Portions borrowed from http://tinyurl.com/2w36o4    *
 *  *
 * ###########################################################  */
#include "router.h"

/* ======================================= */
/* =========    Main Function     ========= */
/* ======================================= */
int main(int argc, char *argv[])
{
int i, j, k, i_i, suffix, sockfd, numbytes;
char ra[2];
char buf[MAXDATASIZE], addr[14], line[81], time_ms_t[17], time_ms[10], src[3], dest[3];
char no_pkts[3];
struct hostent *he;
struct sockaddr_in their_addr;
time_t curtime;
struct timeval tv;
unsigned long long time;
FILE *fp;


if (argc != 3) {
    fprintf(stderr,"usage: injector [traf demand file] [routing algorithm]\n");
    exit(1);
}

sprintf(ra, argv[2], 1);

if ((fp=fopen(argv[1], "r")) == NULL)
    fprintf(stderr,"file not found.\n");

// For each row in the text file, send packets with 0s in header.
//  Fork off children to do this.
for (i=0; i<27; i++)
{
    suffix = 200 - i;
    fgets(line, 82, fp); line[80] = '\0';
    if (!fork()){
     i_i = i;
// Clear the header - but src is same for entire row.
strncpy(&buf[0],"0000000000000000000000000000000000000000", 40);
if (i_i < 10)
sprintf(src,"0%d",i_i);
else
sprintf(src,"%d",i_i);
strncpy(&buf[0],&src[0],2);
  buf[4] = ra[0];
for (j=0; j<27; j++)
{
if (i == j) j++;
if (j == 27) break;
if (j < 10)
sprintf(dest,"0%d",j);
else
sprintf(dest,"%d",j);
strncpy(&buf[2],&dest[0],2);
strncpy(&no_pkts[0],&line[j*3],2); no_pkts[2] = '\0';
```

```
for (k=0; k<atoi(no_pkts); k++)
{
sprintf(addr,"192.168.0.%d", suffix); addr[13] = '\0';
if ((he=gethostbyname(addr)) == NULL)
perror("gethostbyname");
if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) == -1)
perror("socket");
their_addr.sin_family = AF_INET;
their_addr.sin_port = htons(ROUTEPORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);
if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof their_addr) == -1) {
     perror("connect"); exit(1);}
if (send(sockfd, buf, MAXDATASIZE-1, 0) == -1)
    perror("send");
close(sockfd);
printf("sending a packet: %s -> %s\n", src, dest);

}

}
     exit(0);
    }
    else {
    }
}
return 0;
}
```

B.4 file: router.h

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <netdb.h>
#include <sys/time.h>
#include <time.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/ipc.h>

#define ROUTEPORT 3490
#define LOPORT 3491
#define MAXDATASIZE 800000
#define HEADERSIZE 50
#define MAXHOPS 7
#define BACKLOG 10
#define GO_POSX 1
#define GO_POSY 2
#define GO_POSZ 3
#define GO_NEGX 4
#define GO_NEGY 5
#define GO_NEGZ 6
#define FILE_PATH "/pkhome/pkhome/router/routerlogs/"
#define SID_POSX 45
#define SID_POSY 46
#define SID_POSZ 47
#define SID_NEGX 48
#define SID_NEGY 49
#define SID_NEGZ 50
#define MAX_QUEUE 200
#define ADJUSTMENT 2000000
#define LARGENU 1000000
struct queue
{
int val;
unsigned long long lastUpdated;
};
union semun
{
int val;
struct semid_ds *buf;
unsigned short  *array;
struct seminfo  *__buf;
};
```

Appendix C

Appendix C: Matlab Simulation Scripts

C.1 User Interface Scripts

C.1.1 file: do_load.m

```
function do_load();

pattern = input('\n\nEnter a traffic pattern to use... \n0:Flood Traffic (all src/dest pairs)\n1:Tornado Traffic\n
2:Transpose Traffic\n3:Nearest Neighbor\n4:Uniform Random Traffic\n5:Bit-Compliment Traffic\n: ');
option = input('\n\nEnter which routing algorithm to use... \n1:Dimension Ordered\n2:Direction Ordered\n3:Minimal
Oblivious\n4:Minimal Adaptive\n5:Chaos\n6:CQR\n7:Enhanced CQR - Periphery Avoidance\n: ');
k = input('\n\nEnter k...\n: ');
n = input('\n\nEnter n...\n: ');
runs = input('\n\nEnter the number of runs to simulate...\n: ');
output_text_file = 1;
if (option > 3)
    failure = input('\n\nEnter the probability of nodal failures in decimal form...\n: ');
    hs = input('\n\nEnter the percent of nodes which should be hotspots in decimal form...\n: ');
else
    failure = 0.00;
    hs = 0;
end

if (k^n > 50)
    plot = input('\n\nThis seems to be a large set. Would you like to plot these? \n
**This operation could take a while to process!**\n0:No\n1:Yes\n: ');
else
    plot = 1;
end

failed_nodes_queue = 0;
len = k^n;
ch_load = zeros(len, len);
traffic = zeros(len, len);
if(rem(k,2) == 1)
    scale = 3*n*(k/4 - 1/(4*k));  % k even
else
    scale = 6*k*n/4;              % k odd
end

x = traffic/(scale);
traffic = x - diag(diag(x));
```

```
% All src/dest pairs
if(pattern == 0)
    for i=1:k^n
        for j=1:k^n
            traffic(i,j) = runs;
        end
    end

% Tornado Traffic
elseif(pattern == 1)
    for j=1:runs
    for src=1:len
        [x,y,z] = i2dim(src-1, k, n);

        dx = rem(ceil(k/2)-1,k) + x;
        if dx >= k
           dx = dx - k;
        end
        dy = y;
        dz = z;
    dest = dim2i(dx, dy, dz, k, n)+1;
    traffic(src, dest) = traffic(src, dest) + 1;
    end
    end

% Transpose Traffic
elseif(pattern == 2)
    for j=1:runs
    for src=1:len
        [x,y,z] = i2dim(src-1, k, n);
        if(n > 2)
            dest = dim2i(z, x, y, k, n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
            dest = dim2i(z,y,x,k,n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
            dest = dim2i(x,z,y,k,n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
            dest = dim2i(y,x,z,k,n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
            dest = dim2i(y,z,x,k,n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
        else
            z = 0;
            dest = dim2i(y, x, z, k, n)+1;
            traffic(src, dest) = traffic(src,dest) + 1;
        end

    end
    end

% Nearest Neighbor Traffic
elseif(pattern == 3)
    for j=1:runs
    for src=1:len
        [x,y,z] = i2dim(src-1, k, n);
        if(n > 2)
                dest = dim2i(mod(x+1, k), y, z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(mod(x-1, k), y, z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(x, mod(y+1, k), z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(x, mod(y-1, k), z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
```

```
                dest = dim2i(x, y, mod(z+1, k), k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(x, y, mod(z-1, k), k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;

        else
                dest = dim2i(mod(x+1, k), y, z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(mod(x-1, k), y, z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(x, mod(y+1, k), z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
                dest = dim2i(x, mod(y-1, k), z, k, n)+1;
                traffic(src,dest) = traffic(src,dest) + 1;
        end

    end
    end

% Uniform Random Traffic
elseif(pattern == 4)
    for j=1:runs
    for src=1:len
        dest = ceil(rand*len);
        traffic(src,dest) = traffic(src,dest) + ceil(rand()*runs);
    end
    end

% Bit Compliment Traffic
elseif(pattern == 5)
    for j=1:runs
    for src=1:len
        [x,y,z] = i2dim(src-1, k, n);
        if n < 3
            dest = dim2i((k-x-1), (k-y-1), z, k, n)+1;
        else
            dest = dim2i((k-x-1), (k-y-1), (k-z-1), k, n)+1;
        end
        traffic(src,dest) = traffic(src,dest) + 1;
    end
    end
end

if (hs > 0)
    nums=(1:1:(k^n));
    for (j=1:ceil(((k^n)*hs)))
        rand_value = ceil(rand()*length(find(nums>=1)));
        [rows,cols,vals] = find(nums>=1);
        x_ = rows(rand_value);
        y_ = cols(rand_value);
        h_s(j) = nums(x_,y_);
        nums(x_,y_) = 0;
        traffic(:,h_s(j)) = traffic(:,h_s(j))*4;
        [strng] = sprintf('Hotspot Generated at node %d',h_s(j));
        disp(strng);
    end
end

if (output_text_file == 1)
    file_1 = fopen('/Users/lydick/Desktop/file.txt','w');
    for (i=1:k^n)
        fprintf(file_1,'0%d',traffic(i,1));
        for (j=2:k^n)
            if (j == i)
                fprintf(file_1,' 00');
```

```
            else
                fprintf(file_1,' 0%d',traffic(i,j));
            end
        end
        fprintf(file_1,'\n');
    end
    fclose(file_1);
    %traffic
end

% For each traffic dest/source pair
% These are delay independent/load independent visualizations.
%       Mostly this is the visual demand... not the traffic.

if (option <= 3)
    for src=1:len
  for dest=1:len
if(traffic(src,dest) > 0)
           if (option == 1)
                ch_load = ch_load + traffic(src,dest)*dimord_route(src-1, dest-1, k, n);

           elseif (option == 2)
                ch_load = ch_load + traffic(src,dest)*dirord_route(src-1, dest-1, k, n);

           elseif (option == 3)
                ch_load = ch_load + traffic(src,dest)*minobliv_route(src-1, dest-1, k, n);

           end
end
  end
    end
end


elseif (option == 7)
    [ch_load,delay] = do_cqr2(traffic,k,n,failure);
    avg_delay = mean(delay);
    max_delay = max(delay);
    min_delay = min(delay);
    [strng] = sprintf('Mean Delay: \t%0.3g\nMax Delay: \t%d\nMin Delay: \t%d',avg_delay,max_delay,min_delay);
    disp(strng);

elseif (option == 6)
    [ch_load,delay] = do_cqr1(traffic,k,n,failure);
    avg_delay = mean(delay);
    max_delay = max(delay);
    min_delay = min(delay);
    [strng] = sprintf('Mean Delay: \t%0.3g\nMax Delay: \t%d\nMin Delay: \t%d',avg_delay,max_delay,min_delay);
    disp(strng);

elseif (option == 5)
    [ch_load,delay] = do_chaos(traffic,k,n,failure);
    avg_delay = mean(delay);
    max_delay = max(delay);
    min_delay = min(delay);
    [strng] = sprintf('Mean Delay: \t%0.3g\nMax Delay: \t%d\nMin Delay: \t%d',avg_delay,max_delay,min_delay);
    disp(strng);

elseif (option == 4)
    [ch_load,delay,failed_nodes_queue] = do_minadaptive(traffic,k,n,failure);
    avg_delay = mean(delay);
    max_delay = max(delay);
    min_delay = min(delay);
    [strng] = sprintf('Mean Delay: \t%0.3g\nMax Delay: \t%d\nMin Delay: \t%d',avg_delay,max_delay,min_delay);
    disp(strng);
```

```
end

subplot(1,2,1);
max_ch_load = max(max(ch_load));
ch_load = ch_load / max_ch_load;
std_dev = mean(std(ch_load));
if (plot == 1)
    view_load(ch_load, k, n, 1.0-(std_dev), 1.0-(std_dev*0.5));
end


count = 1;
while (count <= (k^n) && plot == 1 && n > 2)
    [val_x,val_y,val_z] = i2dim(count,k,n);
    if (find(failed_nodes_queue == count) >= 1)
        [val1_x,val1_y,val1_z] = sphere(30);
        val1_x = val1_x.*0.05; val1_y = val1_y.*0.05; val1_z = val1_z.*0.05;
        val1_x = val1_x + val_x; val1_y = val1_y + val_y; val1_z = val1_z + val_z;
        plot3(val1_x,val1_y,val1_z,'k');

    elseif ((hs > 0) && (nums(count) == 0))
        [val1_x,val1_y,val1_z] = sphere(30);
        val1_x = val1_x.*0.05; val1_y = val1_y.*0.05; val1_z = val1_z.*0.05;
        val1_x = val1_x + val_x; val1_y = val1_y + val_y; val1_z = val1_z + val_z;
        plot3(val1_x,val1_y,val1_z,'r');

    else
        [val1_x,val1_y,val1_z] = sphere(30);
        val1_x = val1_x.*0.05; val1_y = val1_y.*0.05; val1_z = val1_z.*0.05;
        val1_x = val1_x + val_x; val1_y = val1_y + val_y; val1_z = val1_z + val_z;
        plot3(val1_x,val1_y,val1_z,'b');
    end
    count = count + 1;
end

link_load = zeros(k^n*6,1);
if (n == 3)
    for i=1:k^n
        [x,y,z] = i2dim(i-1,k,n);
        link_load(6*(i-1)+1) = ch_load(i,dim2i(mod(x+1,k),y,z,k,n)+1);
        link_load(6*(i-1)+2) = ch_load(i,dim2i(mod(x-1,k),y,z,k,n)+1);
        link_load(6*(i-1)+3) = ch_load(i,dim2i(x,mod(y+1,k),z,k,n)+1);
        link_load(6*(i-1)+4) = ch_load(i,dim2i(x,mod(y-1,k),z,k,n)+1);
        link_load(6*(i-1)+5) = ch_load(i,dim2i(x,y,mod(z+1,k),k,n)+1);
        link_load(6*(i-1)+6) = ch_load(i,dim2i(x,y,mod(z-1,k),k,n)+1);
    end
else
    [r,c,v] = find(ch_load);
    link_load = v;
end

hold off;
t = [0:1/50:1];
subplot(1,2,2);
[t1,t2] = hist(link_load,t);
hist(link_load,t);
axis([-0.1 1.1 0 max(t1)])
ylabel('Number of Links');
xlabel('Normalized Load per Bi-Directional Link');
title('Link Load Histogram');
legend('Load Distribution');
avg_thpt = mean(link_load);
std_thpt = std(link_load);
[strng] = sprintf('Avg Link Load: \t%2.2f%%\nStd Dev: \t%2.2f', avg_thpt*100,std_thpt*100);
```

```
disp(strng);
error_status = 0;
set(gcf,'Position', [50,500,650,300]);
```

C.1.2 file: view_load.m

```
function view_load(route, k, n, mid, high)
%
%  Use:  view_load(route, k, n, mid, high)
%           load = 2-D matrix containing load info between each pair of
%           nodes
%           k , n = size of torus
%           mid = threshold for coloring loads < mid to green color
%           high = thrshold for coloring loads > high to red color
%               all other loads will be colored yellow
%

len = k^n;
i = 1;
for r=1:len
    for c = 1:len
        if(route(r,c) > 0)
            [sx,sy,sz] = i2dim(r-1, k, n);
            [dx,dy,dz] = i2dim(c-1, k, n);
            if(abs(sx-dx) <= 1)
                x(i,:) = [sx dx];
            else
                x(i,:) = [k-1, k];
            end

            if(abs(sy-dy) <= 1)
                y(i,:) = [sy dy];
            else
                y(i,:) = [k-1, k];
            end

            if(n > 2)
                if(abs(sz-dz) <= 1)
                    z(i,:) = [sz dz];
                else
                    z(i,:) = [k-1, k];
                end
            else
                z(i,:) = [0 0];
            end
            if(route(r,c) < mid)
                color = 'g';
            elseif(route(r,c) > high)
                color = 'r';
            else
                color = 'y';
            end

            if (n > 2)
                plot3(x', y', z', color);

            else
                plot(x',y', color);
            end

            hold on;
            %i = i + 1;
        end
    end
end

grid on;
if (n > 2)
    axis ([0 k 0 k 0 k], 'square');
```

```
    zlabel('z');
else
    axis ([0 k 0 k], 'square');
end

xlabel('x');
ylabel('y');
title('Visualization of Link Utilizations');

```

C.2 Routing Scripts

C.2.1 file: dim2i.m

```
function i = dim2i(x,y,z,k,n)
%
%
%  Use:  function i = dim2i(x, y, z, k, n)
% i = index position
% k = size of k for k-ary n-cube
% n = size of n for k-ary n-cube
%

if(n > 2)
  i = x + k*y + k*k*z;
else
  i = x + k*y;
end
```

C.2.2 file: i2dim.m

```
function [x,varargout] = i2dim(i, k, n)
%
%
%  Use:  function [x,y,z] = index2dim(i, k, n)
% i = index position
% k = size of k for k-ary n-cube
% n = size of n for k-ary n-cube
%
nout = max(nargout,1)-1;
varargout = cell(nout, 1);
x = rem(i,k);
if(n > 1)
  varargout(1) = num2cell(rem(floor(i/k),k));
end

if(n > 2)
  varargout(2) = num2cell(floor(i/(k*k)));
end
```

C.2.3 file: dimord_route.m

```
function route = dimord_route(src, dest, k, n)
%
%   Use:  route = dimord_route(src, dest, k, n)
%

[sx, sy, sz] = i2dim(src, k, n);
[dx, dy, dz] = i2dim(dest, k, n);


len = k^n;
route = zeros(len);

previ = src;

% Route in the x-dimension

currentx = sx;
dirx = dimord(sx, dx, k);
if dirx == -2
    dirx = sign(rand(1)-.5);
end

while currentx ~= dx
    currentx = currentx + dirx;

    if(currentx < 0)
        currentx = currentx + k;
    end
    if(currentx >= k)
        currentx = currentx - k;
    end

    currenti = dim2i(currentx,sy,sz,k,n);
    route(previ+1, currenti+1) = 1;
    previ = currenti;
end

% Route in the y-dimension
if (n > 1)
currenty = sy;
diry = dimord(sy, dy, k);
if diry == -2
    diry = sign(rand(1)-.5);
end

while currenty ~= dy
    currenty = currenty + diry;
    if(currenty < 0)
        currenty = currenty + k;
    end

    if(currenty >= k)
        currenty = currenty - k;
    end

    currenti = dim2i(dx,currenty,sz,k,n);
    route(previ+1, currenti+1) = 1;
    previ = currenti;
end

end % y-dimension

% Route in the z-dimension
```

```
if (n > 2)
currentz = sz;
dirz = dimord(sz, dz, k);
if dirz == -2
    dirz = sign(rand(1)-.5);
end

while currentz ~= dz
    currentz = currentz + dirz;
    if(currentz < 0)
        currentz = currentz + k;
    end
    if(currentz >= k)
        currentz = currentz - k;
    end

    currenti = dim2i(dx, dy, currentz,k,n);
    route(previ+1, currenti+1) = 1;
    previ = currenti;
end

end
```

C.2.4 file: dirord_route.m

```
function route = dirord_route(src, dest, k, n)
%
%   Use:  route = dirord_route(src, dest, k, n)
%               performs direction order routing
%                   (+x, +y, +z, -x, -y, -z)
%

[sx, sy, sz] = i2dim(src, k, n);
[dx, dy, dz] = i2dim(dest, k, n);

len = k^n;
route = zeros(len);

previ = src;

% Get directions in each dimension
currentx = sx;
dirx = dimord(sx, dx, k);
if dirx == -2
    dirx = sign(rand(1)-.5);
end

if (n > 1)
    currenty = sy;
    diry = dimord(sy, dy, k);
    if diry == -2
        diry = sign(rand(1)-.5);
    end
end

if (n > 2)
    currentz = sz;
    dirz = dimord(sz, dz, k);
    if dirz == -2
        dirz = sign(rand(1)-.5);
    end
end


% Route in the +x-dimension
if(dirx > 0)
    while currentx ~= dx
        currentx = currentx + dirx;

        if(currentx < 0)
            currentx = currentx + k;
        end
        if(currentx >= k)
            currentx = currentx - k;
        end

        currenti = dim2i(currentx,sy,sz,k,n);
        route(previ+1, currenti+1) = 1;
        previ = currenti;
    end
end

% Route in the +y-dimension
if(n > 1)
    if(diry > 0)
        while currenty ~= dy
            currenty = currenty + diry;
            if(currenty < 0)
```

```
                currenty = currenty + k;
            end

            if(currenty >= k)
                currenty = currenty - k;
            end

            currenti = dim2i(currentx,currenty,sz,k,n);
            route(previ+1, currenti+1) = 1;
            previ = currenti;
        end
    end
end


% Route in the +z-dimension
if(n > 2)
    if(dirz > 0)
        while currentz ~= dz
            currentz = currentz + dirz;
            if(currentz < 0)
                currentz = currentz + k;
            end
            if(currentz >= k)
                currentz = currentz - k;
            end

            currenti = dim2i(currentx, currenty, currentz,k,n);
            route(previ+1, currenti+1) = 1;
            previ = currenti;
        end
    end
end

% Route in the -x-dimension
if(dirx < 0)
    while currentx ~= dx
        currentx = currentx + dirx;

        if(currentx < 0)
            currentx = currentx + k;
        end
        if(currentx >= k)
            currentx = currentx - k;
        end

        currenti = dim2i(currentx, currenty, currentz,k,n);
        route(previ+1, currenti+1) = 1;
        previ = currenti;
    end
end

% Route in the -y-dimension
if(n > 1)
    if(diry < 0)
        while currenty ~= dy
            currenty = currenty + diry;
            if(currenty < 0)
                currenty = currenty + k;
            end

            if(currenty >= k)
                currenty = currenty - k;
            end

```

```
            currenti = dim2i(currentx, currenty, currentz,k,n);
            route(previ+1, currenti+1) = 1;
            previ = currenti;
        end
    end
end

% Route in the -z-dimension
if(n > 2)
    if(dirz < 0)
        while currentz ~= dz
            currentz = currentz + dirz;
            if(currentz < 0)
                currentz = currentz + k;
            end
            if(currentz >= k)
                currentz = currentz - k;
            end

            currenti = dim2i(currentx, currenty, currentz,k,n);
            route(previ+1, currenti+1) = 1;
            previ = currenti;
        end
    end
end
```

C.2.5 file: minobliv_route.m

```
function route = minobliv_route(src, dest, k, n)
%
%   Use:  route = minobliv_route(src, dest, k, n)
%               performs minimum oblivious routing
%
%

len = k^n;
route = zeros(len);

previ = src;

% Get coordinates of source & destination
[sx, sy, sz] = i2dim(src, k, n);
[dx, dy, dz] = i2dim(dest, k, n);

[dirx, deltax] = dimord(sx, dx, k);

if(n > 1)
    [diry, deltay] = dimord(sy, dy, k);
end
if(n > 2)
    [dirz, deltaz] = dimord(sz, dz, k);
end

% Calculate an intermediate node
% First find the bounding box
ix = round(deltax*rand(1) + sx);

if(ix < 0)
    ix = ix + k;
end
if(ix >= k)
    ix = ix - k;
end

if(n > 1)
    iy = round(deltay*rand(1) + sy);
    if(iy < 0)
        iy = iy + k;
    end
    if(iy >= k)
        iy = iy - k;
    end

end

if(n > 2)
    iz = round(deltaz*rand(1) + sz);
    if(iz < 0)
        iz = iz + k;
    end
    if(iz >= k)
        iz = iz - k;
    end
end

inter_node = dim2i(ix,iy,iz,k,n);
route = dimord_rand_route(src, inter_node, k, n);
route = route + dimord_rand_route(inter_node, dest, k, n);
```

C.2.6 file: do_minadaptive.m

```
function [ch_load, delay, failed_nodes_queue] = do_minadaptive(traffic, k, n, loss);

% Use: [ch_load, delay, failed_nodes_queue] = do_chaos(traffic, k, n, loss);

size = k^n;

threshold = 1;
flit_size = 0.1;
total_deflections = 0;
dropped = 0;

% start the queue with a value which should never occur. Pi.
failed_nodes_queue = pi;

total_original_traffic = 0;

ch_load = zeros(size,size);
delay = [1:size];
for (i=0:size-1)
    delay(i+1) = 0;
end

failure = delay;

total_original_traffic = sum(sum(traffic>0));

for (i=0:size-1)
    if (rand() <= loss)
        failure(i+1) = 1;
        [str,err] = sprintf('Nodal Failure Simulated at node: %d',i);
        disp(str);
        failed_nodes_queue = push(failed_nodes_queue, i);
        dropped = dropped + sum(traffic(i+1,:)>0);
        for (j=1:size)
            traffic(i+1,j) = 0;
        end
        dropped = dropped + sum(traffic(:,i+1)>0);
        for (j=1:size)
            traffic(j,i+1) = 0;
        end

    end
end

total_traffic = 0;

for (i=1:size)
    for (j=1:size)
        total_traffic = total_traffic + traffic(i,j);
    end
end


% While there is traffic within the temp_traffic matrix....

while (length(find(traffic>=1))~=0)

    temp_ch_load = zeros(size,size);
    temp_traffic = traffic;
    [strng,err] = sprintf('...traffic demand of %d packets', length(find(traffic)));
    disp(strng);

    while (length(find(temp_traffic>=1))~=0)
```

```
        % for one node/src pair, randomly select src/dest pair
        rand_value = ceil(rand()*length(find(temp_traffic>=1)));
        [rows,cols,vals] = find(temp_traffic>=1);
        src = rows(rand_value);
        dest = cols(rand_value);


        % Get coordinates for those values
        [sx, sy, sz] = i2dim(src-1, k, n);
        [dx, dy, dz] = i2dim(dest-1, k, n);

        [dirx, deltax] = dimord(sx, dx, k);

        if(n > 1)
            [diry, deltay] = dimord(sy, dy, k);
            dirz = 0;

        end
        if(n > 2)
            [dirz, deltaz] = dimord(sz, dz, k);

        end

        % these print the directions, but before selecting which direction
        % to take, we need to consider that that path may be not as
        % described... I don't think I'm doing this correctly... arg!!

        % Next hop coordinates first equal the source location.
        nx = sx;
        ny = sy;
        nz = sz;
        test_x = -1;
        test_y = -1;
        test_z = -1;
        val_test_x = Inf;
        val_test_y = Inf;
        val_test_z = Inf;
        flag_misroute = 0;
        route_direction = 'x';

        % Test locations, needing to see which temp_traffic values are
        % smallest

        if (dirx ~= 0)
            test_x = dim2i(mod(nx+dirx,k),ny,nz,k,n)+1;
            val_test_x = temp_ch_load(src,test_x);
            if (failure(test_x) == 1)
                val_test_x = Inf;
            end
        end

        if (diry ~= 0)
            test_y = dim2i(nx,mod(ny+diry,k),nz,k,n)+1;
            val_test_y = temp_ch_load(src,test_y);
            if (failure(test_y) == 1)
                val_test_y = Inf;
            end
        end

        if (dirz ~= 0)
            test_z = dim2i(nx,ny,mod(nz+dirz,k),k,n)+1;
            val_test_z = temp_ch_load(src,test_z);
            if (failure(test_z) == 1)
                val_test_z = Inf;
```

```
            end
        end

        flag_no_progress = 0;
        min_val = min(val_test_y, min(val_test_z, val_test_x));
        if (val_test_z == Inf && val_test_y == Inf && val_test_x == Inf && src ~= dest)
            flag_no_progress = 1;
        end

        % If the min val will go over the threshold, we must misroute.
        % First recalculate the minimum of the opposite direction.
        if (min_val + flit_size > threshold && flag_no_progress == 0)
            flag_misroute = 1;
            total_deflections = total_deflections + 1;
            traffic(src,dest) = traffic(src,dest) + 1;
        elseif (flag_no_progress == 1)
            dropped = dropped + 1;
        else

        % If all are the same, this randomizes the direction to route.
            switch(ceil(rand()*3))
                case (1)
                    switch(min_val)
                        case (val_test_x)
                            route_direction = 'x';
                            next_hop = test_x;
                        case (val_test_y)
                            route_direction = 'y';
                            next_hop = test_y;
                        case (val_test_z)
                            route_direction = 'z';
                            next_hop = test_z;
                    end
                case (2)
                    switch(min_val)
                        case (val_test_z)
                            route_direction = 'z';
                            next_hop = test_z;
                        case (val_test_x)
                            route_direction = 'x';
                            next_hop = test_x;
                        case (val_test_y)
                            route_direction = 'y';
                            next_hop = test_y;
                    end
                case (3)
                    switch(min_val)
                        case (val_test_y)
                            route_direction = 'y';
                            next_hop = test_y;
                        case (val_test_z)
                            route_direction = 'z';
                            next_hop = test_z;
                        case (val_test_x)
                            route_direction = 'x';
                            next_hop = test_x;
                    end

            end

        end

        if (flag_misroute == 1)
            delay(src) = delay(src) + 1;
        end
```

```
        temp_traffic(src,dest) = temp_traffic(src,dest) - 1;
        traffic(src,dest) = traffic(src,dest) - 1;
        if (src ~= next_hop && flag_misroute == 0 && flag_no_progress == 0)
            traffic(next_hop,dest) = traffic(next_hop,dest) + 1;
            temp_ch_load(src,next_hop) = temp_ch_load(src,next_hop) + flit_size;
        end



    end

% This gets us to every src/dest going one hop.
ch_load = ch_load + temp_ch_load;
for (i=1:size)
    traffic(i,i) = 0;
end

end

[strng, err] = sprintf('Total deflections: %d', total_deflections);
disp(strng);
[strng, err] = sprintf('Total number of dropped packets from failed nodes: %d', dropped);
disp(strng);
[strng, err] = sprintf('Total traffic demand: %d', total_original_traffic);
disp(strng);
[strng, err] = sprintf('Percent of dropped traffic: %0.03g', dropped/total_original_traffic*100);
disp(strng);
```

C.2.7 file: do_cqr1.m

```
function [ch_load, delay] = do_cqr1(traffic, k, n, loss)

% Use: [ch_load, delay] = do_cqr1(traffic, k, n, loss);
size                = k^n;
threshold           = 100;
flit_size           = 0.1;
total_deflections   = 0;
dropped             = 0;
time = 0;
ch_load             = zeros(size,size);
delay               = zeros(size,1);
failure             = zeros(size,1);

% Calculate and simulate nodal failures.
for i=0:size-1
    if (rand() <= loss)
        failure(i+1) = 1;
        [str,err] = sprintf('Nodal Failure Simulated at node: %d',i);
        disp(str);
        disp(err);
        for j=1:size
            traffic(i+1,j) = 0;
            traffic(j,i+1) = 0;
        end
    end
end

% get rid of any traffic src=dest
for i=1:k^n
    traffic(i,i) = 0;
end

% Find total traffic demand
total_traffic = sum(sum(traffic));

temp_traffic = traffic; % preserve the original traffic matrix.
packet_matrix = cqr_transformTrafficMatrix(temp_traffic); % convert to the packet matrix
extra_temp_traffic = temp_traffic;
packet_queues = zeros(k^n,6);

while(length(find(temp_traffic>=1))~=0)

    %Find a random packet, remove it, change the parameters, and put it
    %back into the matrix.
    [src,dest] = cqr_getRandomValue(temp_traffic);
    [xdir,ydir,zdir,packet_queues] = cqr_chooseQuadrant(src,dest,n,k,packet_queues);
    [dx,dy,dz] = i2dim(dest-1,k,n);
    [sx,sy,sz] = i2dim(src-1,k,n);
    [d_x,deltax] = dimord(sx,dx,k);
    [d_y,deltay] = dimord(sy,dy,k);
    [d_z,deltaz] = dimord(sz,dz,k);
    time = time + abs(deltax) + abs(deltay) + abs(deltaz);
    if (sign(d_x) ~= sign(xdir)) delay(src) = delay(src) + (k-(abs(deltax)+abs(deltax))); end
    if (sign(d_y) ~= sign(ydir)) delay(src) = delay(src) + (k-(abs(deltay)+abs(deltay))); end
    if (sign(d_z) ~= sign(zdir)) delay(src) = delay(src) + (k-(abs(deltaz)+abs(deltaz))); end
    [pk1,packet_matrix,extra_temp_traffic] = cqr_removePacket(packet_matrix,src,dest,extra_temp_traffic);
    pk1.initialized = 1;
    pk1.x_dir = xdir;
    pk1.y_dir = ydir;
    pk1.z_dir = zdir;
    [packet_matrix,extra_temp_traffic] = cqr_addPacket(pk1,packet_matrix,src,dest,extra_temp_traffic);
    temp_traffic(src,dest) = temp_traffic(src,dest) - 1;
end
```

```
% we should now have fully transformed and calculated packet/traffic
% matrices    .

while (length(find(traffic>=1))~=0)

    temp_ch_load = zeros(size,size);
    temp_traffic = traffic;
    temp_packet_matrix = packet_matrix;

    while (length(find(temp_traffic>=1))~=0)

        % for one node/src pair, randomly select src/dest pair
        [src,dest] = cqr_getRandomValue(temp_traffic);

        % Get coordinates for those values
        [pkt,temp_packet_matrix,temp_traffic] = cqr_removePacket(temp_packet_matrix,src,dest,temp_traffic);
        [pkt_global,packet_matrix,traffic] = cqr_removePacket(packet_matrix,src,dest,traffic);

        dirx = pkt.x_dir;
        diry = pkt.y_dir;
        dirz = pkt.z_dir;

        [sx, sy, sz] = i2dim(src-1, k, n);

        % Next hop coordinates first equal the source location.
        nx = sx;
        ny = sy;
        nz = sz;
        test_x = -1;
        test_y = -1;
        test_z = -1;
        val_test_x = Inf;
        val_test_y = Inf;
        val_test_z = Inf;
        flag_misroute = 0;
        flag_no_progress = 0;

        % Test locations, needing to see which temp_traffic values are
        % smallest
        if (dirx ~= 0)
            test_x = dim2i(mod(nx+dirx,k),ny,nz,k,n)+1;
            val_test_x = temp_ch_load(src,test_x);
            if (failure(test_x) == 1)
                val_test_x = Inf;
            end
        end

        if (diry ~= 0)
            test_y = dim2i(nx,mod(ny+diry,k),nz,k,n)+1;
            val_test_y = temp_ch_load(src,test_y);
            if (failure(test_y) == 1)
                val_test_y = Inf;
            end
        end

        if (dirz ~= 0)
            test_z = dim2i(nx,ny,mod(nz+dirz,k),k,n)+1;
            val_test_z = temp_ch_load(src,test_z);
            if (failure(test_z) == 1)
                val_test_z = Inf;
            end
        end

        min_val = min(val_test_y, min(val_test_z, val_test_x));
```

```
        if (val_test_z == Inf && val_test_y == Inf && val_test_x == Inf)
            if (src ~= dest)
                flag_no_progress = 1;
                [strng] = sprintf('%d-->%d :: xdir:%d  ydir:%d  zdir:%d  TIME:%d',src,dest,dirx,diry,dirz,time);
                disp(strng);
            else
                %packet reached its destination... DROP IT!
            end
        end

        % If the min val will go over the threshold, we must misroute.
        % First recalculate the minimum of the opposite direction.
        if (min_val + flit_size > threshold && flag_no_progress == 0)
            flag_misroute = 1;
            total_deflections = total_deflections + 1;
            delay(src) = delay(src) + 1;
            % we queue back only globally - not locally. There is nothing
            % we can do until the next iteration with the threshold being
            % surpassed.
            [packet_matrix,traffic] = cqr_addPacket(pkt_global,packet_matrix,src,dest,traffic);

        elseif (flag_no_progress == 1 && src ~= dest)
            dropped = dropped + 1;
        elseif (src == dest)
            disp('packet reached destination');
        else

        % If all are the same, this randomizes the direction to route.
            switch(ceil(rand()*3))
                case (1)
                    switch(min_val)
                        case (val_test_x)
                            next_hop = test_x;
                        case (val_test_y)
                            next_hop = test_y;
                        case (val_test_z)
                            next_hop = test_z;
                    end
                case (2)
                    switch(min_val)
                        case (val_test_z)
                            next_hop = test_z;
                        case (val_test_x)
                            next_hop = test_x;
                        case (val_test_y)
                            next_hop = test_y;
                    end
                case (3)
                    switch(min_val)
                        case (val_test_y)
                            next_hop = test_y;
                        case (val_test_z)
                            next_hop = test_z;
                        case (val_test_x)
                            next_hop = test_x;
                    end

            end

          % we now have our next_hop
          [nx, ny, nz] = i2dim(next_hop-1, k, n);
          [dx, dy, dz] = i2dim(dest-1, k, n);
          d_x = dimord(nx,dx,k);
          d_y = dimord(ny,dy,k);
```

```
            d_z = dimord(nz,dz,k);
            pkt_global.x_dir = abs(pkt_global.x_dir) * d_x;
            pkt_global.y_dir = abs(pkt_global.y_dir) * d_y;
            pkt_global.z_dir = abs(pkt_global.z_dir) * d_z;

            z_dir_temp = pkt_global.z_dir;
            if (n < 3)
                z_dir_temp = 0;
            end

            if (pkt_global.x_dir == 0 && pkt_global.y_dir == 0 && z_dir_temp == 0)
            else
                [packet_matrix, traffic] = cqr_addPacket(pkt_global,packet_matrix,next_hop,dest,traffic);
            end
            temp_ch_load(src,next_hop) = temp_ch_load(src,next_hop) + flit_size;
        end
    end

% This gets us to every src/dest going one hop.
ch_load = ch_load + temp_ch_load;
for i=1:size
    traffic(i,i) = 0;
end

end

[strng, err] = sprintf('Total deflections: %d', total_deflections);
disp(strng);
[strng, err] = sprintf('Total number of dropped packets from failed nodes: %d', dropped);
disp(strng);
[strng, err] = sprintf('Total traffic demand: %d', total_traffic);
disp(strng);
[strng, err] = sprintf('Percent of dropped traffic: %0.03g', dropped/total_traffic*100);
disp(strng);
[strng, err] = sprintf('Total number of hops: %d', time);
disp(strng);
```

C.2.8 file: do_cqr2.m

```
function [ch_load, delay] = do_cqr2(traffic, k, n, loss)
% Use: [ch_load, delay] = do_cqr2(traffic, k, n, loss);

size                = k^n;
threshold           = 100;
flit_size           = 0.1;
total_deflections   = 0;
dropped             = 0;
time = 0;
ch_load             = zeros(size,size);
delay               = zeros(size,1);
failure             = zeros(size,1);

% Calculate and simulate nodal failures.
for i=0:size-1
    if (rand() <= loss)
        failure(i+1) = 1;
        [str,err] = sprintf('Nodal Failure Simulated at node: %d',i);
        disp(str);
        disp(err);
        for j=1:size
            traffic(i+1,j) = 0;
            traffic(j,i+1) = 0;
        end
    end
end

% get rid of any traffic src=dest
for i=1:k^n
    traffic(i,i) = 0;
end

% Find total traffic demand
total_traffic = sum(sum(traffic));

temp_traffic = traffic; % preserve the original traffic matrix.
packet_matrix = cqr_transformTrafficMatrix(temp_traffic); % convert to the packet matrix
extra_temp_traffic = temp_traffic;
packet_queues = zeros(k^n,6);

while(length(find(temp_traffic>=1))~=0)

    %Find a random packet, remove it, change the parameters, and put it
    %back into the matrix.
    [src,dest] = cqr_getRandomValue(temp_traffic);
    [xdir,ydir,zdir,packet_queues] = cqr_chooseQuadrant_v(src,dest,n,k,packet_queues);
    [pk1,packet_matrix,extra_temp_traffic] = cqr_removePacket(packet_matrix,src,dest,extra_temp_traffic);
    [dx,dy,dz] = i2dim(dest-1,k,n);
    [sx,sy,sz] = i2dim(src-1,k,n);
    [d_x,deltax] = dimord(sx,dx,k);
    [d_y,deltay] = dimord(sy,dy,k);
    [d_z,deltaz] = dimord(sz,dz,k);
    if (sign(d_x) ~= sign(xdir)) delay(src) = delay(src) + (k-(abs(deltax)+abs(deltax))); end
    if (sign(d_y) ~= sign(ydir)) delay(src) = delay(src) + (k-(abs(deltay)+abs(deltay))); end
    if (sign(d_z) ~= sign(zdir)) delay(src) = delay(src) + (k-(abs(deltaz)+abs(deltaz))); end
    pk1.initialized = 1;
    pk1.x_dir = xdir;
    pk1.y_dir = ydir;
    pk1.z_dir = zdir;
    [packet_matrix,extra_temp_traffic] = cqr_addPacket(pk1,packet_matrix,src,dest,extra_temp_traffic);
    temp_traffic(src,dest) = temp_traffic(src,dest) - 1;
end

```

```
% we should now have fully transformed and calculated packet/traffic
% matrices    .


while (length(find(traffic>=1))~=0)

    temp_ch_load = zeros(size,size);
    temp_traffic = traffic;
    temp_packet_matrix = packet_matrix;

    while (length(find(temp_traffic>=1))~=0)
        time = time + 1;
        % for one node/src pair, randomly select src/dest pair
        [src,dest] = cqr_getRandomValue(temp_traffic);

        % Get coordinates for those values
        [pkt,temp_packet_matrix,temp_traffic] = cqr_removePacket(temp_packet_matrix,src,dest,temp_traffic);
        [pkt_global,packet_matrix,traffic] = cqr_removePacket(packet_matrix,src,dest,traffic);

        dirx = pkt.x_dir;
        diry = pkt.y_dir;
        dirz = pkt.z_dir;

        [sx, sy, sz] = i2dim(src-1, k, n);

        % Next hop coordinates first equal the source location.
        nx = sx;
        ny = sy;
        nz = sz;
        test_x = -1;
        test_y = -1;
        test_z = -1;
        val_test_x = Inf;
        val_test_y = Inf;
        val_test_z = Inf;
        flag_misroute = 0;
        flag_no_progress = 0;


        % Test locations, needing to see which temp_traffic values are
        % smallest
        delta_total = abs(dirx) + abs(diry) + abs(dirz);

        if (dirx ~= 0)
            test_x = dim2i(mod(nx+sign(dirx),k),ny,nz,k,n)+1;
            val_test_x = (1 + temp_ch_load(src,test_x)) * (1 - (abs(dirx) / delta_total));
            if (failure(test_x) == 1)
                val_test_x = Inf;
            end
        end

        if (diry ~= 0)
            test_y = dim2i(nx,mod(ny+sign(diry),k),nz,k,n)+1;
            val_test_y = (1 + temp_ch_load(src,test_y)) * (1 - (abs(diry) / delta_total));
            if (failure(test_y) == 1)
                val_test_y = Inf;
            end
        end

        if (dirz ~= 0)
            test_z = dim2i(nx,ny,mod(nz+sign(dirz),k),k,n)+1;
            val_test_z = (1 + temp_ch_load(src,test_z)) * (1 - (abs(dirz) / delta_total));
            if (failure(test_z) == 1)
                val_test_z = Inf;
            end
```

```
        end

        min_val = min(val_test_y, min(val_test_z, val_test_x));

        if (val_test_z == Inf && val_test_y == Inf && val_test_x == Inf)
            if (src ~= dest)
                flag_no_progress = 1;
                [strng] = sprintf('packet cannot move(?!) %d-->%d :: xdir:%d  ydir:%d  zdir:%d  TIME:%d',src,dest,dirx,diry,dirz
                disp(strng);
            else
                %packet reached its destination... DROP IT!
            end
        end

        % If the min val will go over the threshold, we must misroute.
        % First recalculate the minimum of the opposite direction.
        if (min_val + flit_size > threshold && flag_no_progress == 0)
            flag_misroute = 1;
            total_deflections = total_deflections + 1;
            delay(src) = delay(src) + 1;
            % we queue back only globally - not locally. There is nothing
            % we can do until the next iteration with the threshold being
            % surpassed.
            [packet_matrix,traffic] = cqr_addPacket(pkt_global,packet_matrix,src,dest,traffic);

        elseif (flag_no_progress == 1 && src ~= dest)
            dropped = dropped + 1;
        elseif (src == dest)
            disp('packet reached destination');
        else

            d_x1 = pkt_global.x_dir;
            d_y1 = pkt_global.y_dir;
            d_z1 = pkt_global.z_dir;
        % If all are the same, this randomizes the direction to route.
            switch(ceil(rand()*3))
                case (1)
                    switch(min_val)
                        case (val_test_x)
                            next_hop = test_x;
                            d_x1 = (sign(pkt_global.x_dir) * -1)  + pkt_global.x_dir;
                        case (val_test_y)
                            next_hop = test_y;
                            d_y1 = (sign(pkt_global.y_dir) * -1)  + pkt_global.y_dir;
                        case (val_test_z)
                            next_hop = test_z;
                            d_z1 = (sign(pkt_global.z_dir) * -1)  + pkt_global.z_dir;
                    end
                case (2)
                    switch(min_val)
                        case (val_test_z)
                            next_hop = test_z;
                            d_z1 = (sign(pkt_global.z_dir) * -1)  + pkt_global.z_dir;
                        case (val_test_x)
                            next_hop = test_x;
                            d_x1 = (sign(pkt_global.x_dir) * -1)  + pkt_global.x_dir;
                        case (val_test_y)
                            next_hop = test_y;
                            d_y1 = (sign(pkt_global.y_dir) * -1)  + pkt_global.y_dir;
                    end
                case (3)
                    switch(min_val)
                        case (val_test_y)
                            next_hop = test_y;
                            d_y1 = (sign(pkt_global.y_dir) * -1)  + pkt_global.y_dir;
```

```
                        case (val_test_z)
                            next_hop = test_z;
                            d_z1 = (sign(pkt_global.z_dir) * -1)  + pkt_global.z_dir;
                        case (val_test_x)
                            next_hop = test_x;
                            d_x1 = (sign(pkt_global.x_dir) * -1)  + pkt_global.x_dir;
                    end

            end

          % we now have our next_hop
          [nx, ny, nz] = i2dim(next_hop-1, k, n);
          [dx, dy, dz] = i2dim(dest-1, k, n);
          pkt_global.x_dir = d_x1;
          pkt_global.y_dir = d_y1;
          pkt_global.z_dir = d_z1;
          z_dir_temp = pkt_global.z_dir;
          if (n < 3)
              z_dir_temp = 0;
          end

          if (pkt_global.x_dir == 0 && pkt_global.y_dir == 0 && z_dir_temp == 0)
              %do nothing, drop it, it's successfully reached the dest.
          else
              [packet_matrix, traffic] = cqr_addPacket(pkt_global,packet_matrix,next_hop,dest,traffic);
          end
          temp_ch_load(src,next_hop) = temp_ch_load(src,next_hop) + flit_size;

        end
  end

% This gets us to every src/dest going one hop.
ch_load = ch_load + temp_ch_load;
for i=1:size
    traffic(i,i) = 0;
end

end

[strng, err] = sprintf('Total deflections: %d', total_deflections);
disp(strng);
[strng, err] = sprintf('Total number of dropped packets from failed nodes: %d', dropped);
disp(strng);
[strng, err] = sprintf('Total traffic demand: %d', total_traffic);
disp(strng);
[strng, err] = sprintf('Percent of dropped traffic: %0.03g', dropped/total_traffic*100);
disp(strng);
```

C.2.9 file: cqr_addPacket.m

```
function [packet_matrix, traffic_matrix] = cqr_addPacket(packet, packet_matrix_in, src, dest, traffic_matrix_in)

traffic_matrix_in(src,dest) = traffic_matrix_in(src,dest) + 1;
val = traffic_matrix_in(src,dest);

packet_matrix_in(src,dest,val).valid = packet.valid;
packet_matrix_in(src,dest,val).initialized = packet.initialized;
packet_matrix_in(src,dest,val).x_dir = packet.x_dir;
packet_matrix_in(src,dest,val).y_dir = packet.y_dir;
packet_matrix_in(src,dest,val).z_dir = packet.z_dir;

packet_matrix = packet_matrix_in;
traffic_matrix = traffic_matrix_in;
```

C.2.10 file: cqr_chooseQuadrant.m

```
function [dir_x,dir_y,dir_z,newqueues] = cqr_chooseQuadrant(src,dest,n,k,queues)
% Use: [x_dir,y_dir,z_dir,newqueues] = cqr_chooseQuadrant(src,dest,n,k,queues)

    queue_x_pos = queues(src,1);
    queue_x_neg = queues(src,2);
    queue_y_pos = queues(src,3);
    queue_y_neg = queues(src,4);
    queue_z_pos = queues(src,5);
    queue_z_neg = queues(src,6);

    [sx, sy, sz] = i2dim(src-1, k, n);
    [dx, dy, dz] = i2dim(dest-1, k, n);




    if (n == 3)
        % for quadrant I: (+,+,+)
        [dir_q1x,q1_deltax] = dimord(sx, dx, k);
        [dir_q1y,q1_deltay] = dimord(sy, dy, k);
        [dir_q1z,q1_deltaz] = dimord(sz, dz, k);
        nu_of_queues = 3;
        if (sign(dir_q1x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q1x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q1y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q1y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q1z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q1x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q1_val = (abs(q1_deltax) + abs(q1_deltay) + abs(q1_deltaz)) * (t_q/nu_of_queues);
        else q1_val = Inf;
        end

        min_val = q1_val;
        dir_x = sign(q1_deltax);
        dir_y = sign(q1_deltay);
        dir_z = sign(q1_deltaz);


        % for quadrant II: (+ + -)
        [dir_q2x,q2_deltax] = dimord(sx, dx, k);
        [dir_q2y,q2_deltay] = dimord(sy, dy, k);
        [dir_q2z,q2_deltaz] = dimord(sz, dz, k);
        if (q2_deltaz > 0) q2_deltaz = q2_deltaz-k;
        elseif (q2_deltaz < 0) q2_deltaz = q2_deltaz+k;
```

```
        else q2_deltaz = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q2x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q2x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q2y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q2y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q2z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q2x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q2_val = (abs(q2_deltax) + abs(q2_deltay) + abs(q2_deltaz)) * (t_q/nu_of_queues);
        else q2_val = Inf;
        end

        if (q2_val < min_val)
            min_val = q2_val;
            dir_x = sign(q2_deltax);
            dir_y = sign(q2_deltay);
            dir_z = sign(q2_deltaz);
        end


        % for quadrant III: (+ - +)
        [dir_q3x,q3_deltax] = dimord(sx, dx, k);
        [dir_q3y,q3_deltay] = dimord(sy, dy, k);
        [dir_q3z,q3_deltaz] = dimord(sz, dz, k);
        if (q3_deltay > 0) q3_deltay = q3_deltay-k;
        elseif (q3_deltay < 0) q3_deltay = q3_deltay+k;
        else q3_deltay = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q3x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q3x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q3y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q3y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
```

```
        if (sign(dir_q3z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q3x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q3_val = (abs(q3_deltax) + abs(q3_deltay) + abs(q3_deltaz)) * (t_q/nu_of_queues);
        else q3_val = Inf;
        end

        if (q3_val < min_val)
            min_val = q3_val;
            dir_x = sign(q3_deltax);
            dir_y = sign(q3_deltay);
            dir_z = sign(q3_deltaz);
        end


        % for quadrant IV: (+ - -)
        [dir_q4x,q4_deltax] = dimord(sx, dx, k);
        [dir_q4y,q4_deltay] = dimord(sy, dy, k);
        [dir_q4z,q4_deltaz] = dimord(sz, dz, k);
        if (q4_deltay > 0) q4_deltay = q4_deltay-k;
        elseif (q4_deltay < 0) q4_deltay = q4_deltay+k;
        else q4_deltay = 0;
        end
        if (q4_deltaz > 0) q4_deltaz = q4_deltaz-k;
        elseif (q4_deltaz < 0) q4_deltaz = q4_deltaz+k;
        else q4_deltaz = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q4x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q4x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q4y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q4y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q4z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q4x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q4_val = (abs(q4_deltax) + abs(q4_deltay) + abs(q4_deltaz)) * (t_q/nu_of_queues);
        else q4_val = Inf;
        end

        if (q4_val < min_val)
            min_val = q4_val;
```

```
            dir_x = sign(q4_deltax);
            dir_y = sign(q4_deltay);
            dir_z = sign(q4_deltaz);
        end


        % for quadrant V: (- + +)
        [dir_q5x,q5_deltax] = dimord(sx, dx, k);
        [dir_q5y,q5_deltay] = dimord(sy, dy, k);
        [dir_q5z,q5_deltaz] = dimord(sz, dz, k);
        if (q5_deltax > 0) q5_deltax = q5_deltax-k;
        elseif (q5_deltax < 0) q5_deltax = q5_deltax+k;
        else q5_deltax = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q5x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q5x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q5y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q5y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q5z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q5x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q5_val = (abs(q5_deltax) + abs(q5_deltay) + abs(q5_deltaz)) * (t_q/nu_of_queues);
        else q5_val = Inf;
        end

        if (q5_val < min_val)
            min_val = q5_val;
            dir_x = sign(q5_deltax);
            dir_y = sign(q5_deltay);
            dir_z = sign(q5_deltaz);
        end


        % for quadrant VI: (- + -)
        [dir_q6x,q6_deltax] = dimord(sx, dx, k);
        [dir_q6y,q6_deltay] = dimord(sy, dy, k);
        [dir_q6z,q6_deltaz] = dimord(sz, dz, k);
        if (q6_deltax > 0) q6_deltax = q6_deltax-k;
        elseif (q6_deltax < 0) q6_deltax = q6_deltax+k;
        else q6_deltax = 0;
        end
        if (q6_deltaz > 0) q6_deltaz = q6_deltaz-k;
        elseif (q6_deltaz < 0) q6_deltaz = q6_deltaz+k;
        else q6_deltaz = 0;
        end
        nu_of_queues = 3;
```

```
        if (sign(dir_q6x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q6x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q6y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q6y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q6z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q6x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q6_val = (abs(q6_deltax) + abs(q6_deltay) + abs(q6_deltaz)) * (t_q/nu_of_queues);
        else q6_val = Inf;
        end

        if (q6_val < min_val)
            min_val = q6_val;
            dir_x = sign(q6_deltax);
            dir_y = sign(q6_deltay);
            dir_z = sign(q6_deltaz);
        end


        % for quadrant VII: (- - +)
        [dir_q7x,q7_deltax] = dimord(sx, dx, k);
        [dir_q7y,q7_deltay] = dimord(sy, dy, k);
        [dir_q7z,q7_deltaz] = dimord(sz, dz, k);
        if (q7_deltax > 0) q7_deltax = q7_deltax-k;
        elseif (q7_deltax < 0) q7_deltax = q7_deltax+k;
        else q7_deltax = 0;
        end
        if (q7_deltay > 0) q7_deltay = q7_deltay-k;
        elseif (q7_deltay < 0) q7_deltay = q7_deltay+k;
        else q7_deltay = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q7x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q7x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q7y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q7y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
```

```
        end

        if (sign(dir_q7z) < 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q7x) > 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q7_val = (abs(q7_deltax) + abs(q7_deltay) + abs(q7_deltaz)) * (t_q/nu_of_queues);
        else q7_val = Inf;
        end

        if (q7_val < min_val)
            min_val = q7_val;
            dir_x = sign(q7_deltax);
            dir_y = sign(q7_deltay);
            dir_z = sign(q7_deltaz);
        end


        % for quadrant VIII: (- - -)
        [dir_q8x,q8_deltax] = dimord(sx, dx, k);
        [dir_q8y,q8_deltay] = dimord(sy, dy, k);
        [dir_q8z,q8_deltaz] = dimord(sz, dz, k);
        if (q8_deltax > 0) q8_deltax = q8_deltax-k;
        elseif (q8_deltax < 0) q8_deltax = q8_deltax+k;
        else q8_deltax = 0;
        end
        if (q8_deltay > 0) q8_deltay = q8_deltay-k;
        elseif (q8_deltay < 0) q8_deltay = q8_deltay+k;
        else q8_deltay = 0;
        end
        if (q8_deltaz > 0) q8_deltaz = q8_deltaz-k;
        elseif (q8_deltaz < 0) q8_deltaz = q8_deltaz+k;
        else q8_deltaz = 0;
        end
        nu_of_queues = 3;
        if (sign(dir_q8x) > 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q8x) < 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q8y) > 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q8y) < 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q8z) > 0)
            t_q = t_q + queue_z_neg;
        elseif (sign(dir_q8x) < 0)
            t_q = t_q + queue_z_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end
        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q8_val = (abs(q8_deltax) + abs(q8_deltay) + abs(q8_deltaz)) * (t_q/nu_of_queues);
```

```
        else q8_val = Inf;
        end

        if (q8_val < min_val)
            min_val = q8_val;
            dir_x = sign(q8_deltax);
            dir_y = sign(q8_deltay);
            dir_z = sign(q8_deltaz);
        end


        else

        % for quadrant I: (+ +)
        [dir_q1x,q1_deltax] = dimord(sx, dx, k);
        [dir_q1y,q1_deltay] = dimord(sy, dy, k);
        nu_of_queues = 2;
        if (sign(dir_q1x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q1x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q1y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q1y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q1_val = (q1_deltax + q1_deltay) * (t_q/nu_of_queues);
        else q1_val = Inf;
        end

        min_val = q1_val;
        dir_x = sign(q1_deltax);
        dir_y = sign(q1_deltay);



        % for quadrant II: (+ -)
        [dir_q2x,q2_deltax] = dimord(sx, dx, k);
        [dir_q2y,q2_deltay] = dimord(sy, dy, k);
        if (q2_deltay > 0) q2_deltay = q2_deltay - k;
        elseif (q2_deltay < 0) q2_deltay = q2_deltay + k;
        else q2_deltay = 0;
        end
        nu_of_queues = 2;
        if (sign(dir_q2x) < 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q2x) > 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q2y) > 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q2y) < 0)
```

```
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q2_val = (abs(q2_deltax) + abs(q2_deltay)) * (t_q/nu_of_queues);
        else q2_val = Inf;
        end

        if (q2_val < min_val)
            min_val = q2_val;
            dir_x = sign(q2_deltax);
            dir_y = sign(q2_deltay);
        end


        % for quadrant III: (- +)
        [dir_q3x,q3_deltax] = dimord(sx, dx, k);
        [dir_q3y,q3_deltay] = dimord(sy, dy, k);
        if (q3_deltax > 0) q3_deltax = q3_deltax - k;
        elseif (q3_deltax < 0) q3_deltax = q3_deltax + k;
        else q3_deltax = 0;
        end
        nu_of_queues = 2;
        if (sign(dir_q3x) > 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q3x) < 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q3y) < 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q3y) > 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q3_val = (abs(q3_deltax) + abs(q3_deltay)) * (t_q/nu_of_queues);
        else q3_val = Inf;
        end

        if (q3_val < min_val)
            min_val = q3_val;
            dir_x = sign(q3_deltax);
            dir_y = sign(q3_deltay);
        end


        % for quadrant IV: (- -)
        [dir_q4x,q4_deltax] = dimord(sx, dx, k);
        [dir_q4y,q4_deltay] = dimord(sy, dy, k);
        if (q4_deltay > 0) q4_deltay = q4_deltay - k;
        elseif (q4_deltay < 0) q4_deltay = q4_deltay + k;
        else q4_deltay = 0;
        end
        if (q4_deltax > 0) q4_deltax = q4_deltax - k;
        elseif (q4_deltax < 0) q4_deltax = q4_deltax + k;
        else q4_deltax = 0;
        end
```

```
        nu_of_queues = 2;
        if (sign(dir_q4x) > 0)
            t_q = queue_x_neg;
        elseif (sign(dir_q4x) < 0)
            t_q = queue_x_pos;
        else
            t_q = 0;
            nu_of_queues = nu_of_queues - 1;
        end

        if (sign(dir_q4y) > 0)
            t_q = t_q + queue_y_neg;
        elseif (sign(dir_q4y) < 0)
            t_q = t_q + queue_y_pos;
        else
            nu_of_queues = nu_of_queues - 1;
        end

        t_q = t_q + 1;
        if (nu_of_queues ~= 0) q4_val = (abs(q4_deltax) + abs(q4_deltay)) * (t_q/nu_of_queues);
        else q4_val = Inf;
        end

        if (q4_val < min_val)
            min_val = q4_val;
            dir_x = sign(q4_deltax);
            dir_y = sign(q4_deltay);
        end

        dir_z = 0;

    end

    newqueues = queues;

    if (sign(dir_x) > 0)
        newqueues(src,1) = queue_x_pos + 1;
    elseif (sign(dir_x) < 0)
        newqueues(src,2) = queue_x_neg + 1;
    end

    if (sign(dir_y) > 0)
        newqueues(src,3) = queue_y_pos + 1;
    elseif (sign(dir_y) < 0)
        newqueues(src,4) = queue_y_neg + 1;
    end

    if (sign(dir_z) > 0)
        newqueues(src,5) = queue_z_pos + 1;
    elseif (sign(dir_z) < 0)
        newqueues(src,6) = queue_z_neg + 1;
    end

```

C.2.11 file: cqr_getRandomValue.m

```
function [src_,dest_,v] = cqr_getRandomValue(traff)
% use: [src,dest,v] = cqr_getRandomValue(traffic_matrix);

    rand_value = ceil(rand()*length(find(traff>=1)));
    [rows,cols,vals] = find(traff>=1);
    src_ = rows(rand_value);
    dest_ = cols(rand_value);
    v = ceil(vals(rand_value)*rand());
```

C.2.12 file: cqr_transformTrafficMatrix.m

```
function [packet_matrix_] = cqr_transformTrafficMatrix (traf_initial)
    %use: [packet_matrix] = cqr_transformTrafficMatrix (traffic_matrix);

    traf = traf_initial;
    size = length(traf);
    for i=1:size
        for j=1:size
            for k = 1:max(max(traf))
                packet_matrix_(i,j,k).initialized = 0;
                packet_matrix_(i,j,k).valid = 0;
                packet_matrix_(i,j,k).x_dir = 0;
                packet_matrix_(i,j,k).y_dir = 0;
                packet_matrix_(i,j,k).z_dir = 0;
            end
        end
    end

    while (length(find(traf>=1))~=0)
        [src,dest,v] = cqr_getRandomValue(traf);
        for i=1:size
            if (packet_matrix_(src,dest,i).valid == 0)
                value = i;
                break;
            end
        end
        packet_matrix_(src,dest,i).valid = 1;
        traf(src,dest) = traf(src,dest) - 1;
    end
```

Appendix D

Appendix D: Full Laboratory Results

D.1 Matlab Plot Scripts

D.1.1 file: convertLoad.m

```
function convertLoad(k,n,matrix);
% usage: convertLoad(k,n,load_matrix)
%       where, load_matrix is the load distribution taken directly
%       from the 'interconnectThroughput' script.

load_matrix = zeros(k^n);

for (i=1:(k^n))
    node = 28-i;
    negy = matrix(node,1);
    posy = matrix(node,2);
    posx = matrix(node,3);
    negx = matrix(node,4);
    negz = matrix(node,5);
    posz = matrix(node,6);
    [x,y,z] = i2dim((i-1),k,n);
    node_posx = dim2i(mod(x+1,k),y,z,k,n) + 1;
    node_negx = dim2i(mod(x-1,k),y,z,k,n) + 1;
    node_posy = dim2i(x,mod(y+1,k),z,k,n) + 1;
    node_negy = dim2i(x,mod(y-1,k),z,k,n) + 1;
    node_posz = dim2i(x,y,mod(z+1,k),k,n) + 1;
    node_negz = dim2i(x,y,mod(z-1,k),k,n) + 1;
    load_matrix(i,node_posx) = posx;
    load_matrix(i,node_negx) = negx;
    load_matrix(i,node_posy) = posy;
    load_matrix(i,node_negy) = negy;
    load_matrix(i,node_posz) = posz;
    load_matrix(i,node_negz) = negz;
end

max_ch_load = max(max(load_matrix));
load_matrix = load_matrix / max_ch_load;
std_dev = mean(std(load_matrix));
subplot(1,2,1);
view_load(load_matrix, k, n, 1.0-(std_dev), 1.0-(std_dev*0.5));

count = 0;

while (count < (k^n))
    [val_x,val_y,val_z] = i2dim(count,k,n);
```

```
    [val1_x,val1_y,val1_z] = sphere(30);
    val1_x = val1_x.*0.05; val1_y = val1_y.*0.05; val1_z = val1_z.*0.05;
    val1_x = val1_x + val_x; val1_y = val1_y + val_y; val1_z = val1_z + val_z;
    plot3(val1_x,val1_y,val1_z,'b');
    count = count + 1;
end
set(gcf,'Position', [50,500,650,300])
```

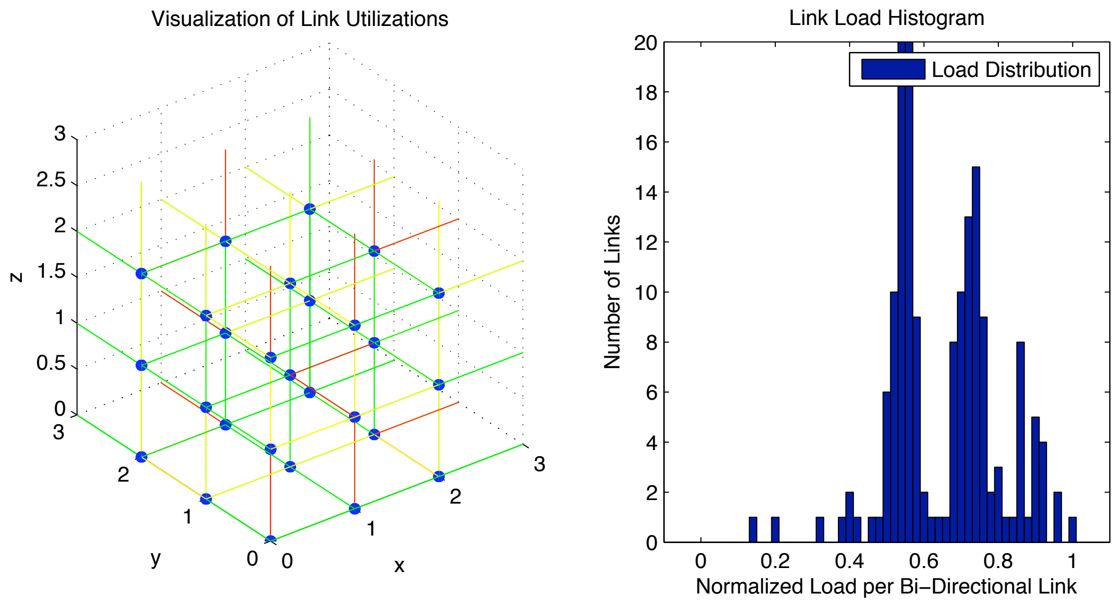


Figure D.1: *Results of Dimension Ordered Routing (DOR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.1.2 file: view_load.m

See Appendix C.1.2.

D.1.3 file: i2dim.m

See Appendix C.2.2.

D.1.4 file: dim2i.m

See Appendix C.2.1.

D.2 Static Algorithms

D.2.1 Dimension Ordered Routing Results

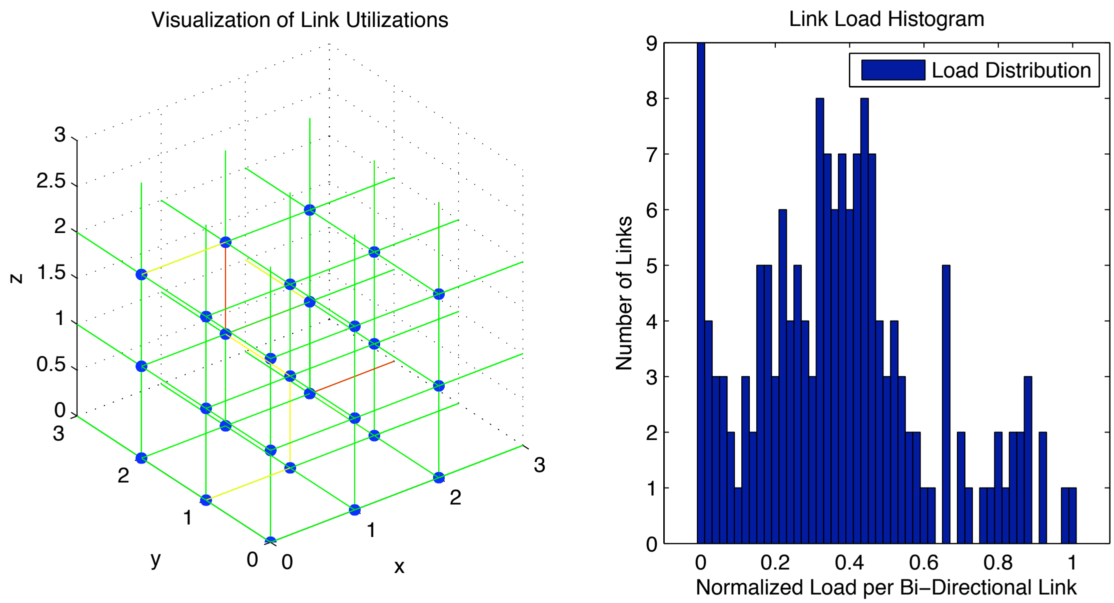


Figure D.2: *Results of Dimension Ordered Routing (DOR) using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

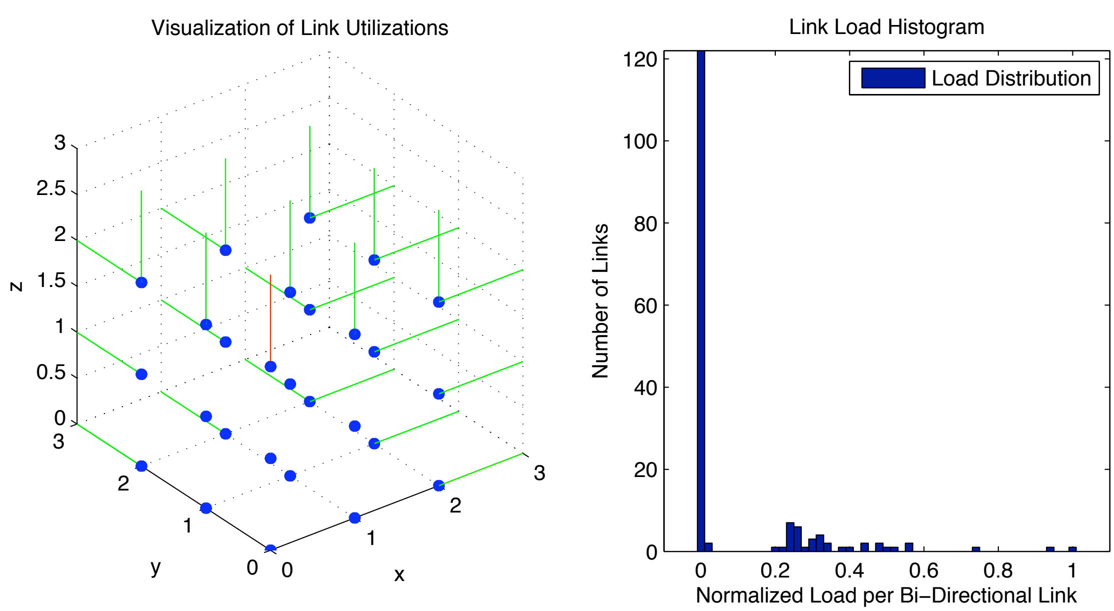


Figure D.3: *Results of Dimension Ordered Routing (DOR) using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

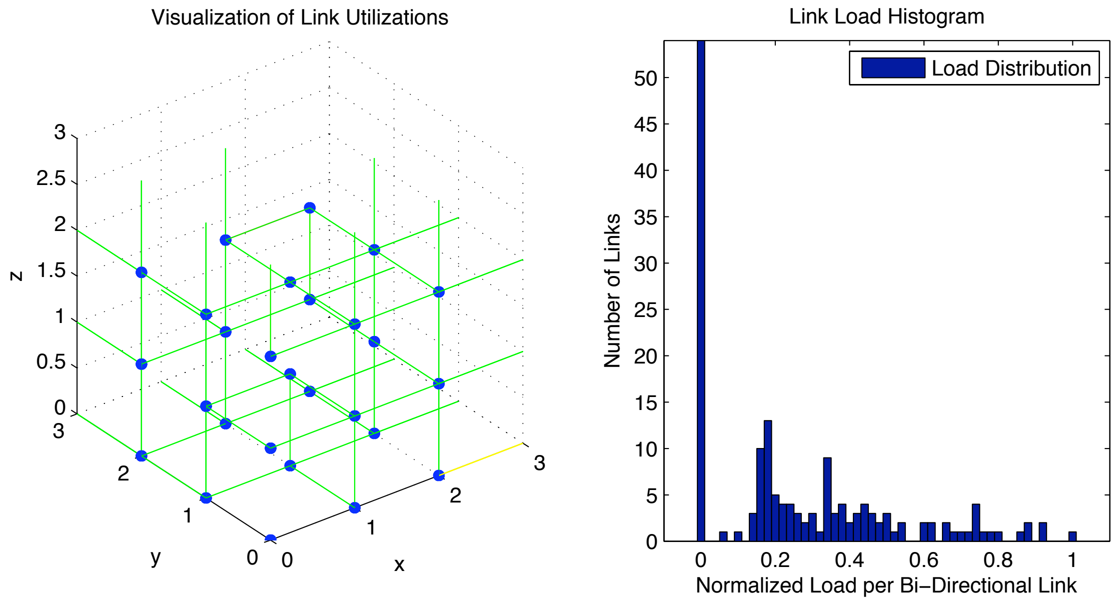


Figure D.4: *Results of Dimension Ordered Routing (DOR) using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

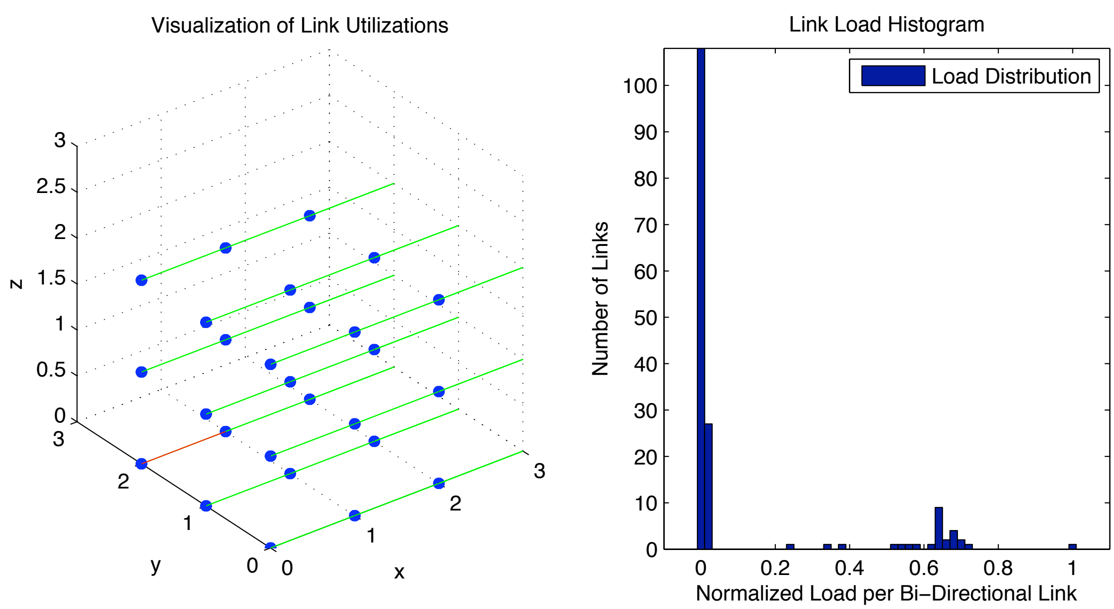


Figure D.5: *Results of Dimension Ordered Routing (DOR) using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

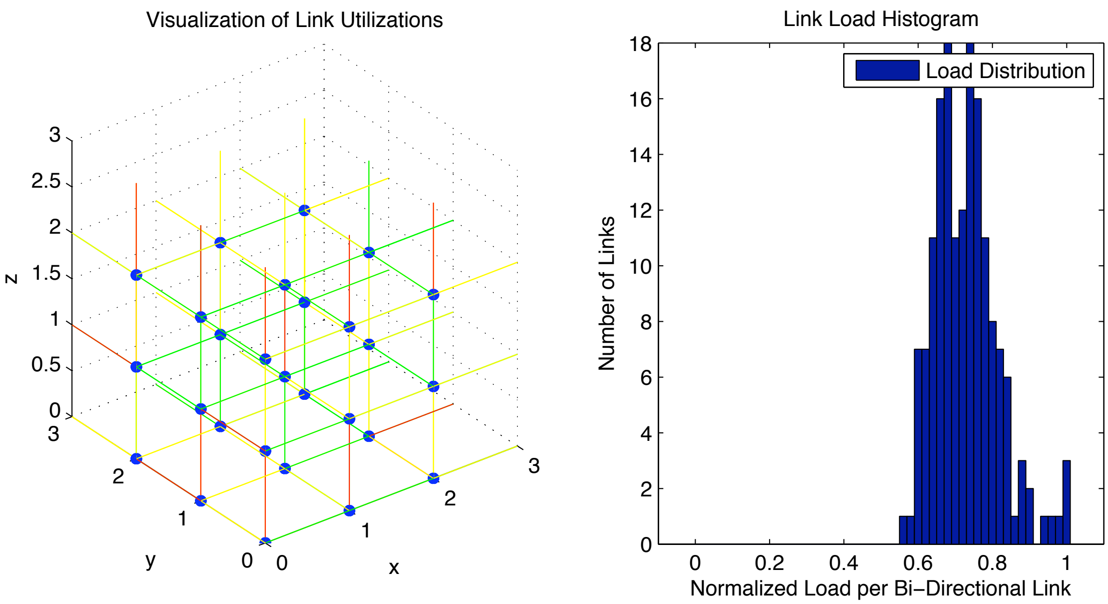


Figure D.6: *Results of Dimension Ordered Routing (DOR) using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

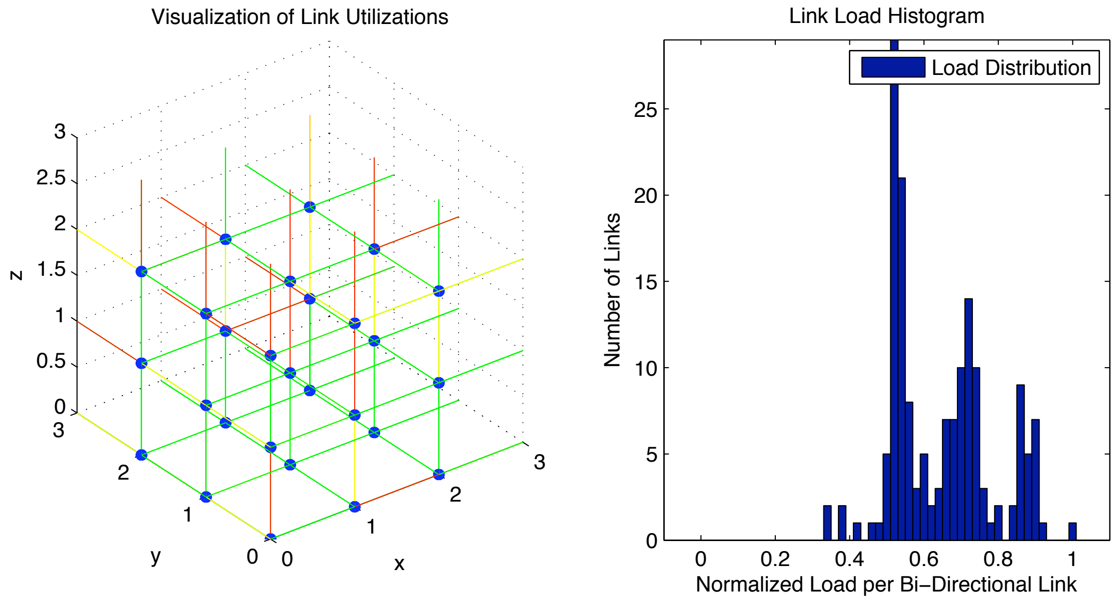


Figure D.7: *Results of Direction Ordered Routing (DIR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.2.2 Direction Ordered Routing Results

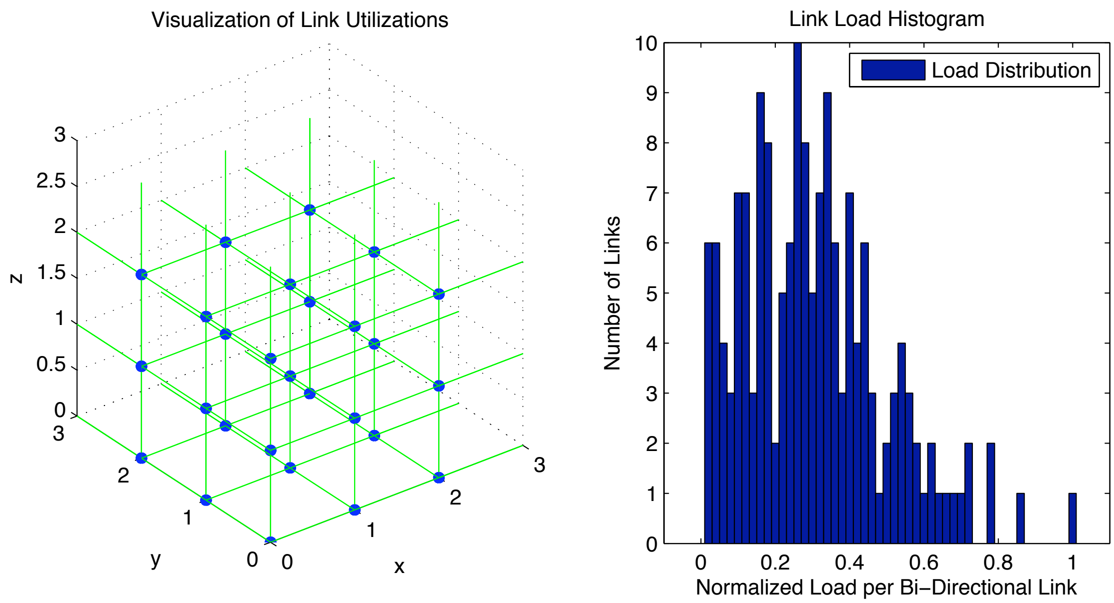


Figure D.8: *Results of Direction Ordered Routing (DIR) using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

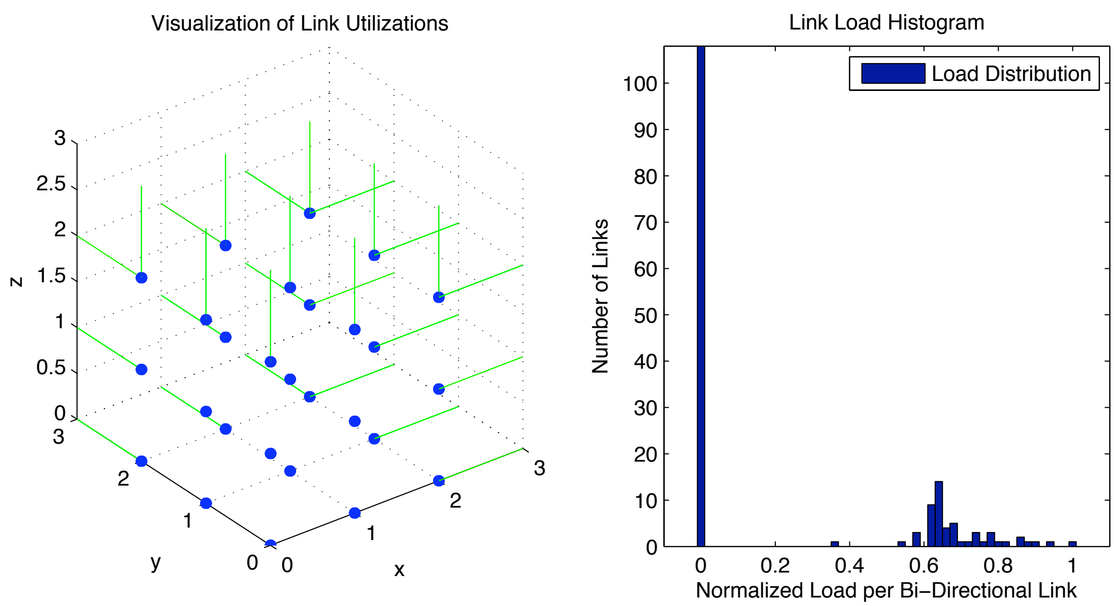


Figure D.9: *Results of Direction Ordered Routing (DIR) using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

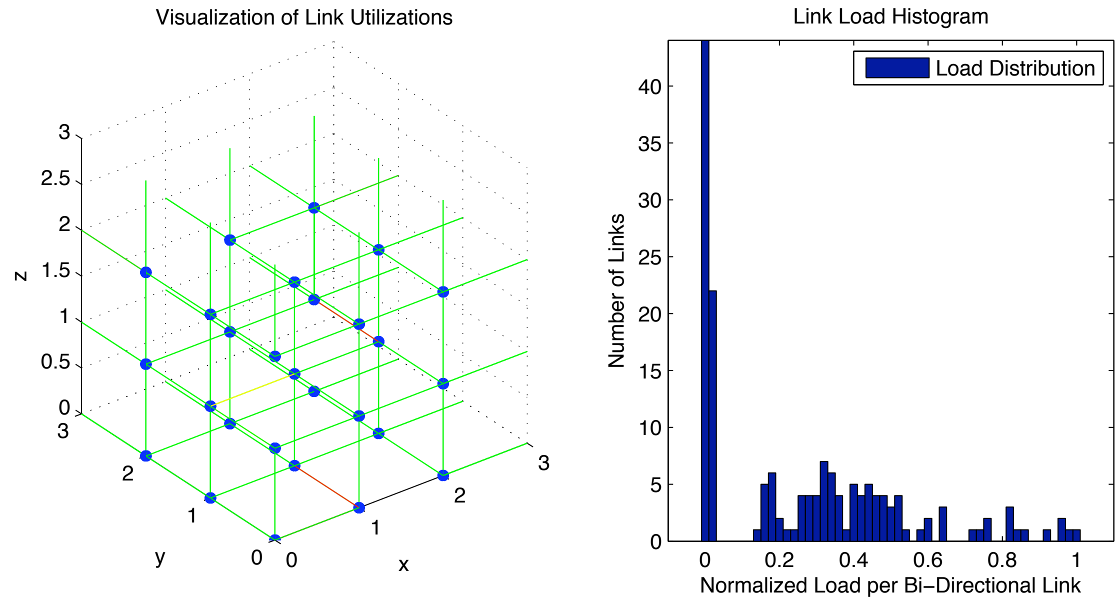


Figure D.10: *Results of Direction Ordered Routing (DIR) using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

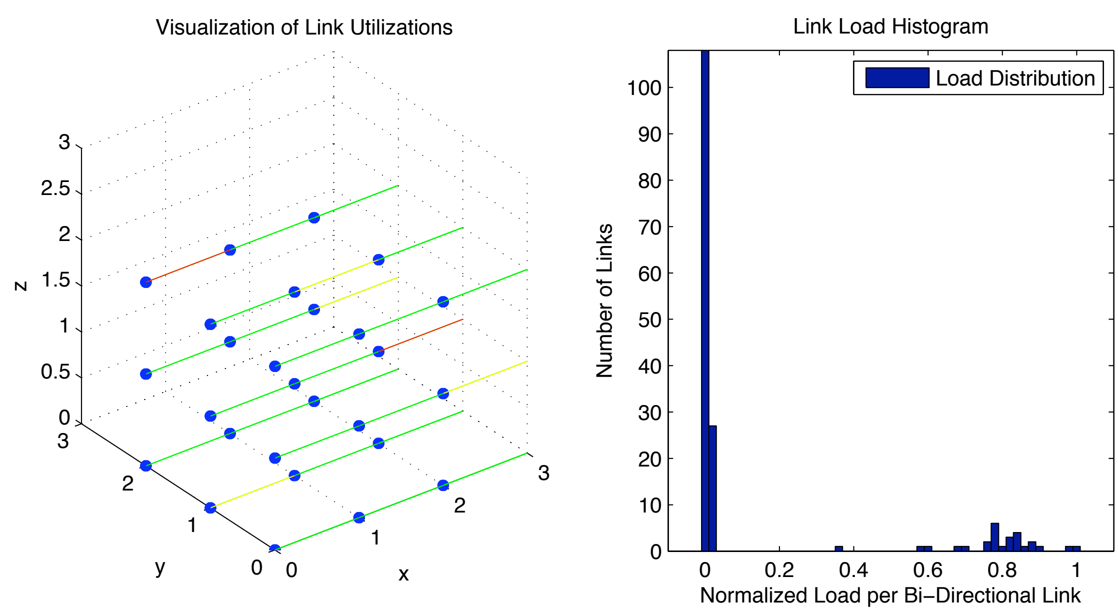


Figure D.11: *Results of Direction Ordered Routing (DIR) using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

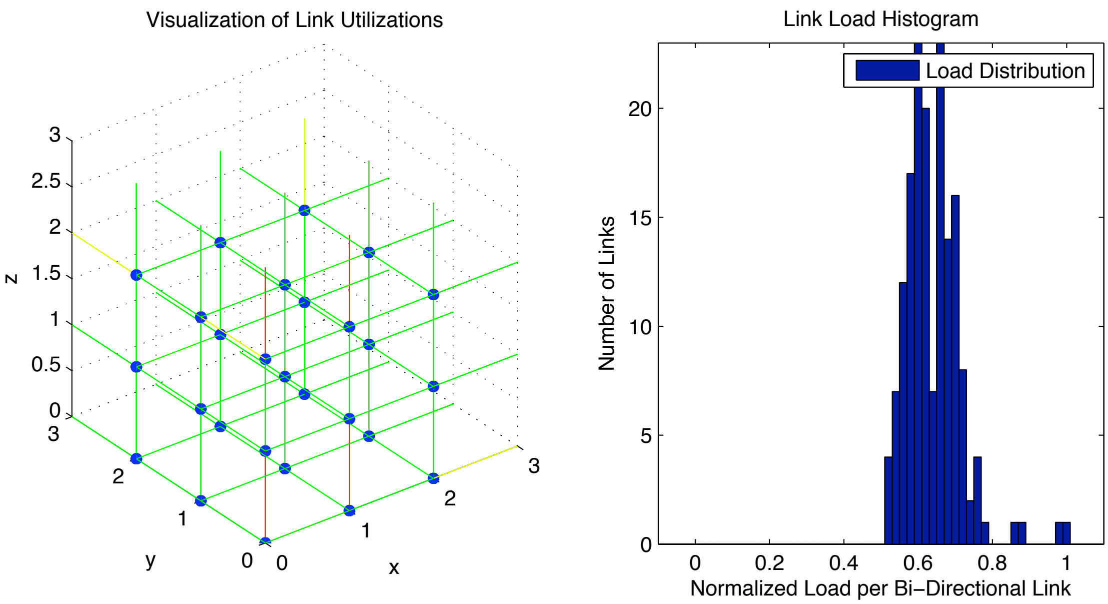


Figure D.12: *Results of Direction Ordered Routing (DIR) using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

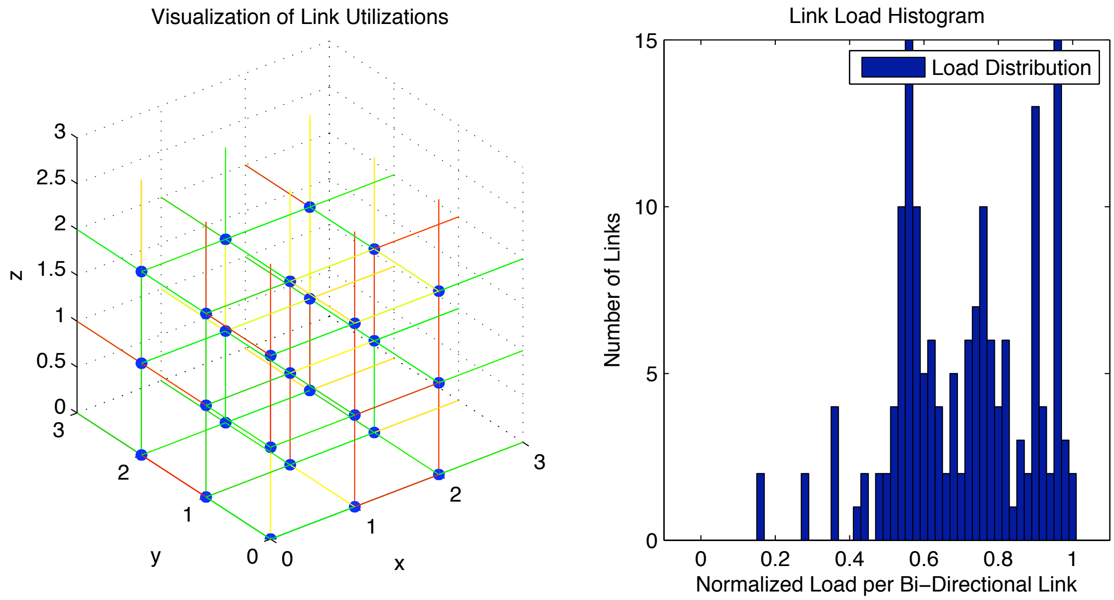


Figure D.13: *Results of Minimal Oblivious Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.3 Oblivious Algorithms

D.3.1 Minimal Oblivious Routing Results

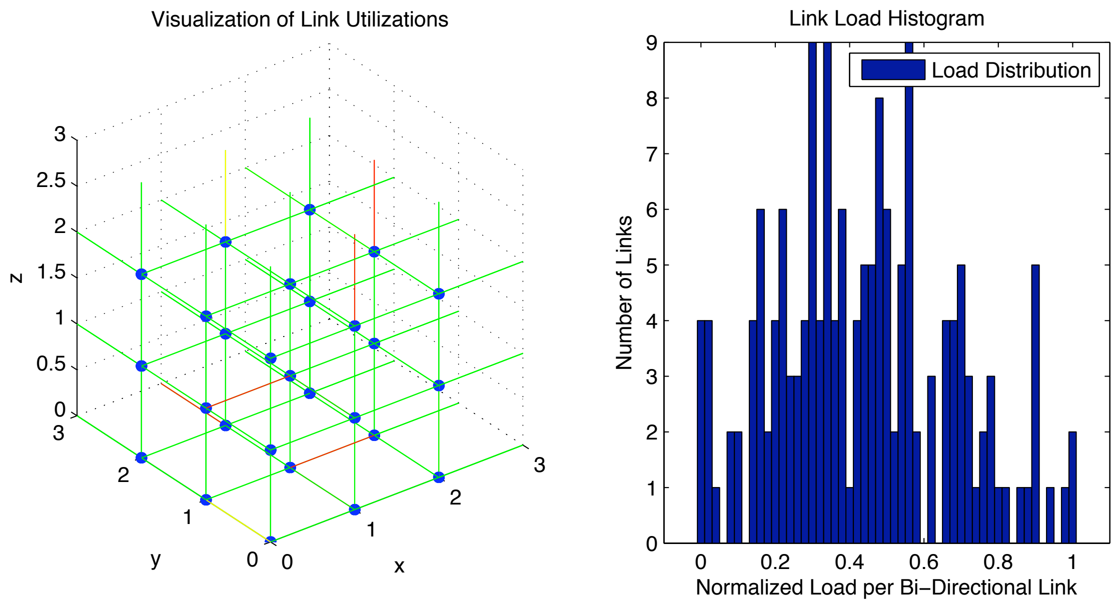


Figure D.14: *Results of Minimal Oblivious Routing using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

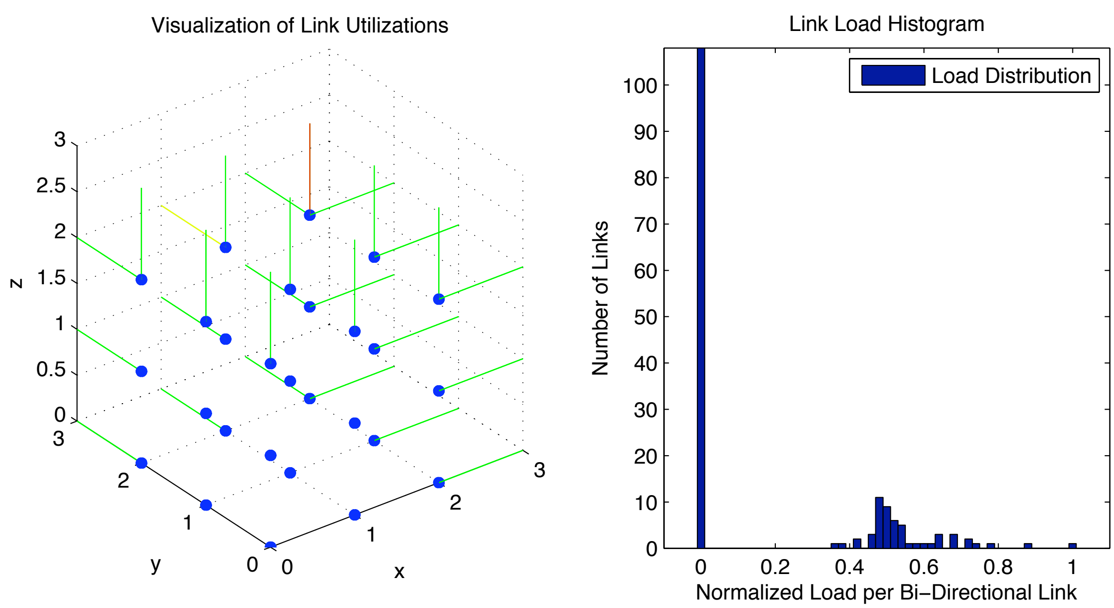


Figure D.15: *Results of Minimal Oblivious Routing using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

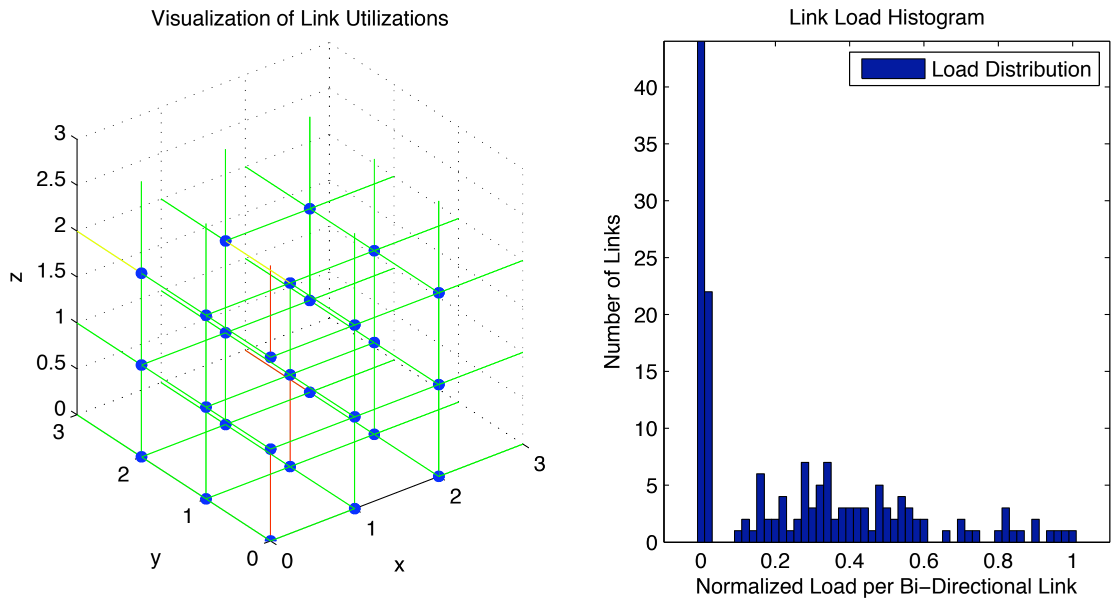


Figure D.16: *Results of Minimal Oblivious Routing using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

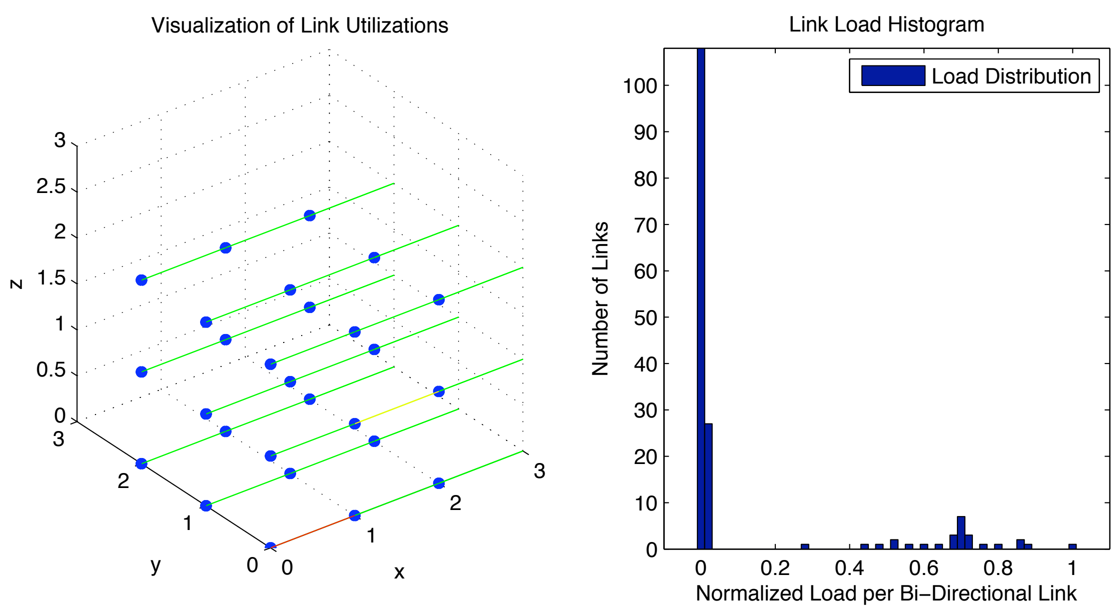


Figure D.17: *Results of Minimal Oblivious Routing using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

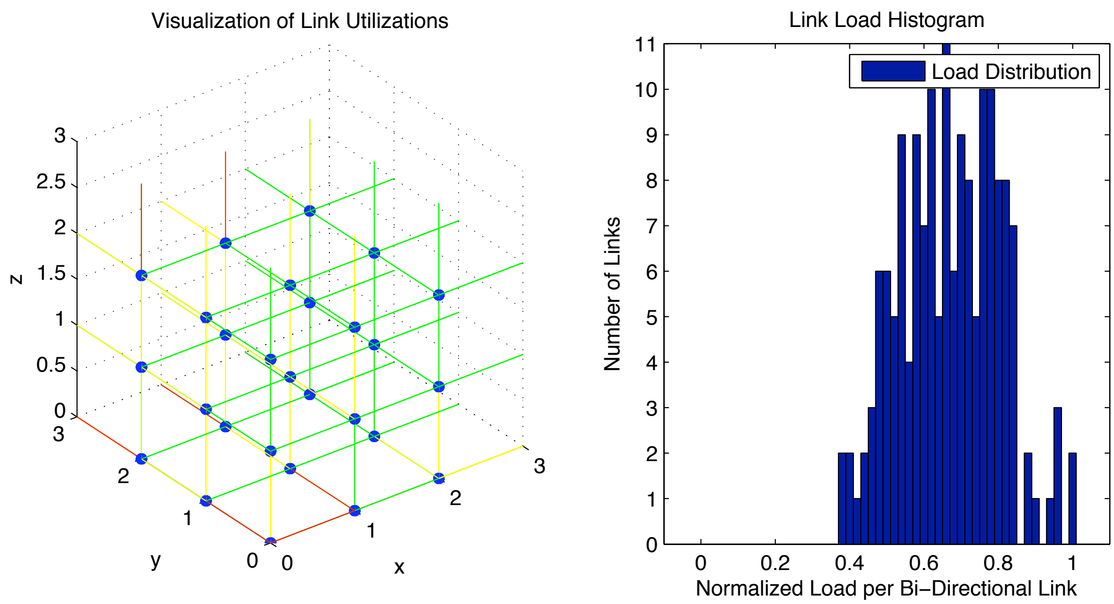


Figure D.18: *Results of Minimal Oblivious Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

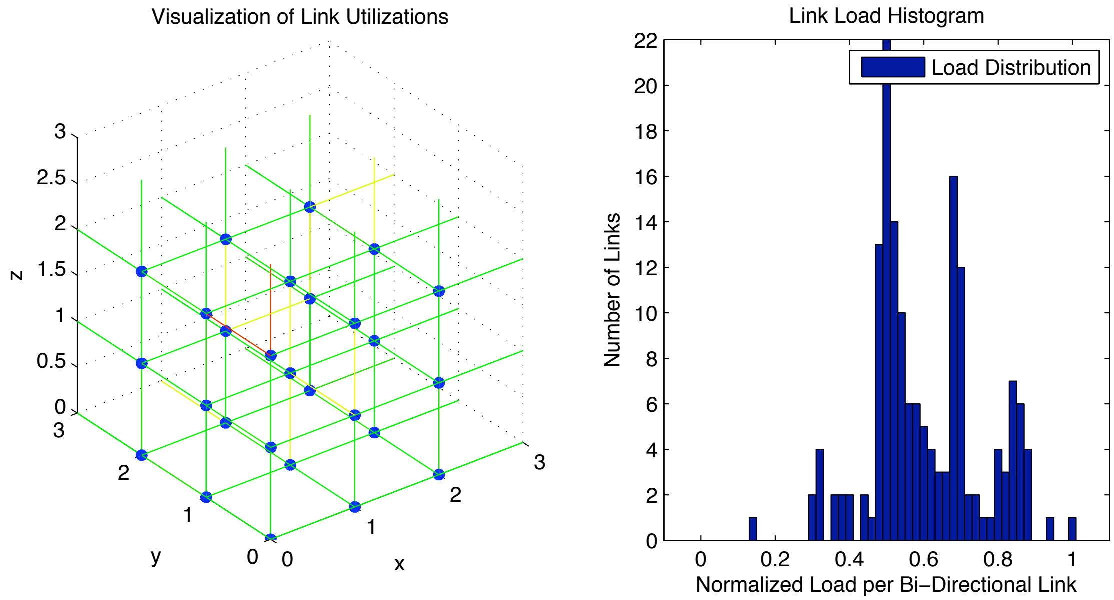


Figure D.19: *Results of Minimal Adaptive Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.4 Adaptive Algorithms

D.4.1 Minimal Adaptive Routing Results

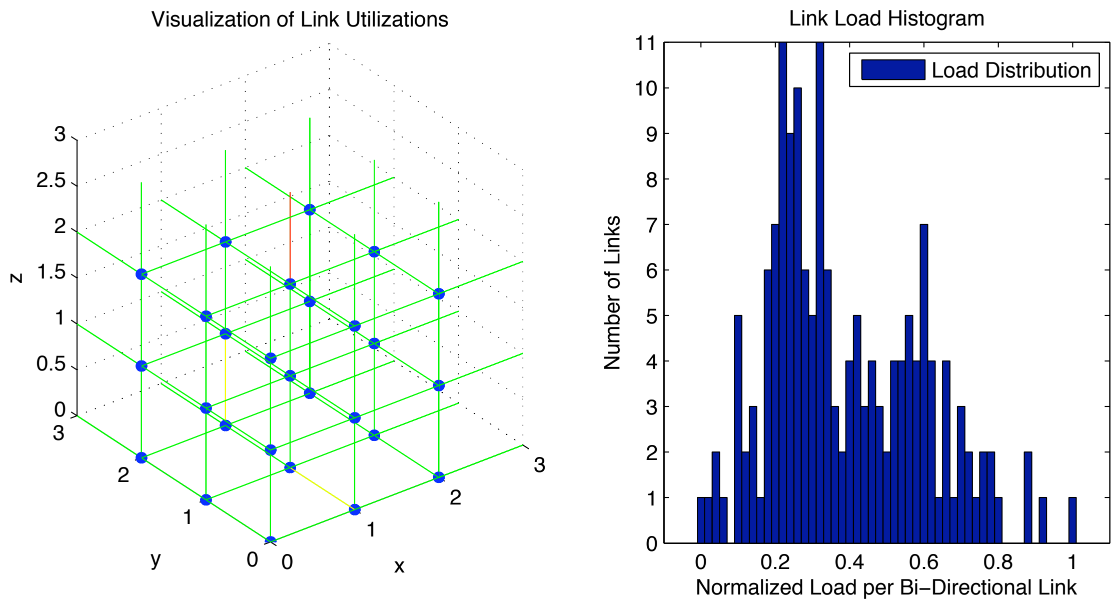


Figure D.20: *Results of Minimal Adaptive Routing using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

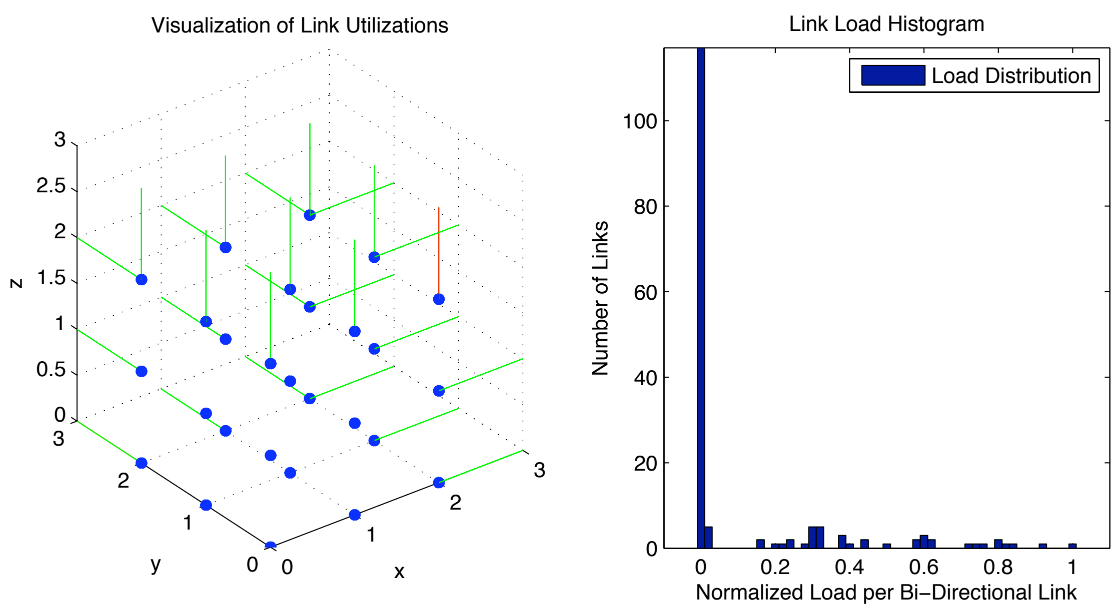


Figure D.21: *Results of Minimal Adaptive Routing using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

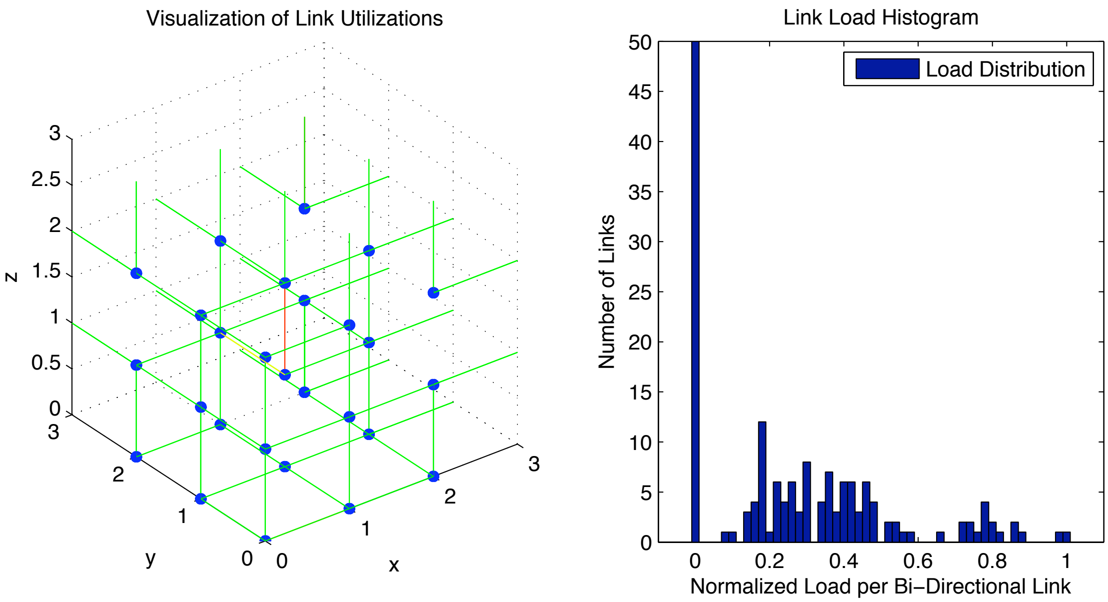


Figure D.22: *Results of Minimal Adaptive Routing using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

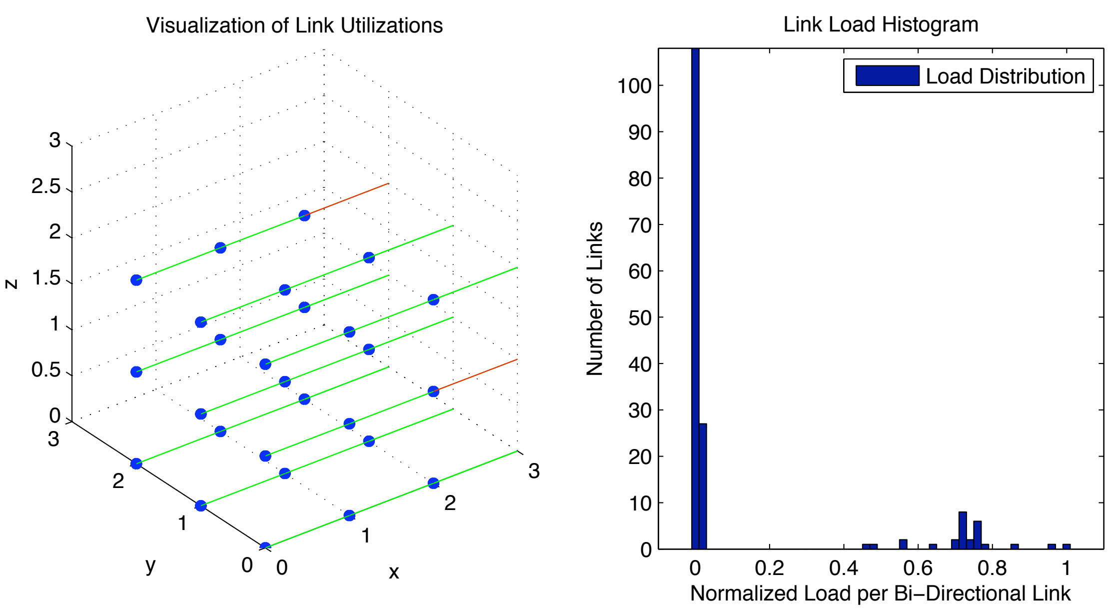


Figure D.23: *Results of Minimal Adaptive Routing using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

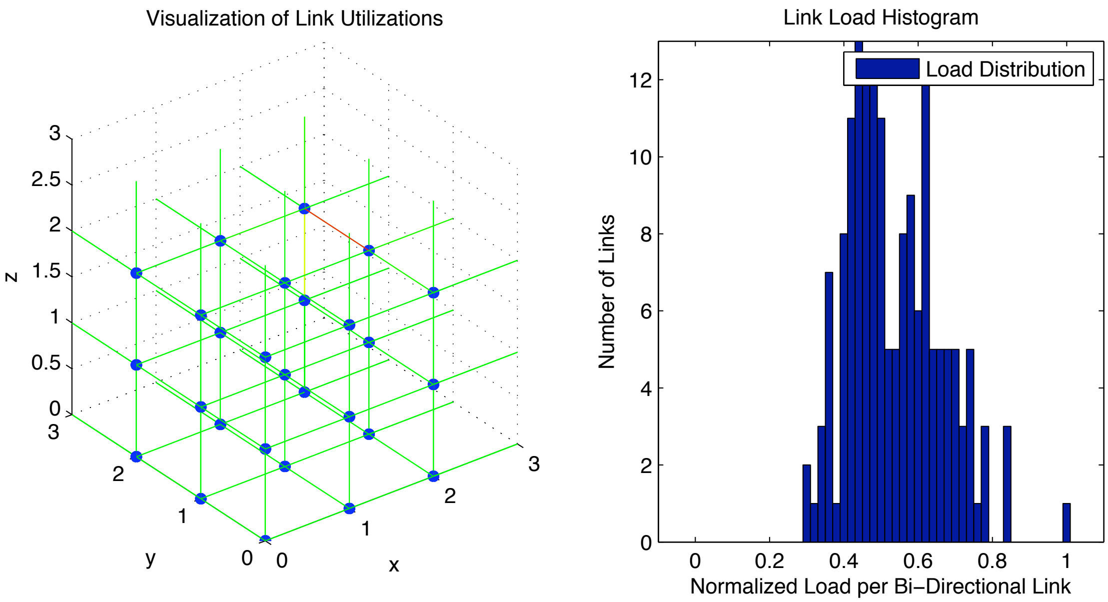


Figure D.24: *Results of Minimal Adaptive Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

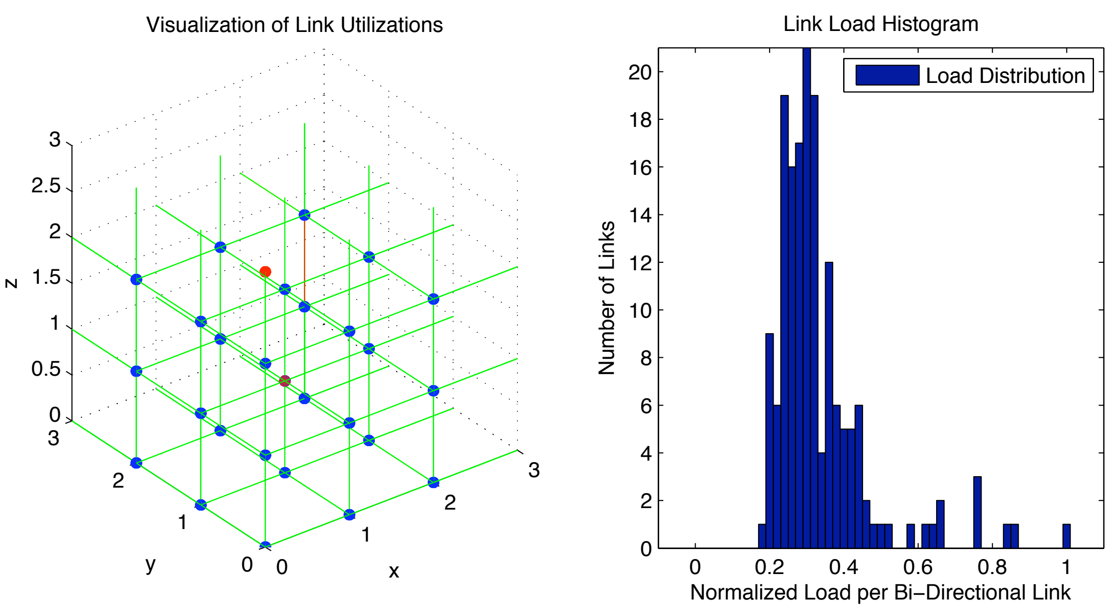


Figure D.25: *Results of Minimal Adaptive Routing using a flood traffic pattern (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

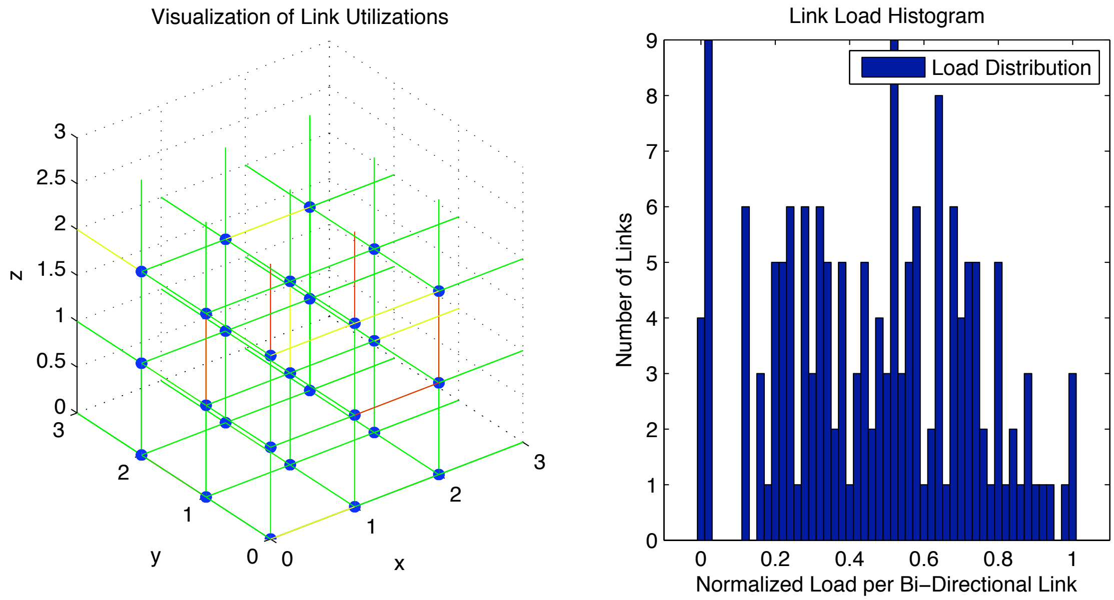


Figure D.26: *Results of CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.4.2 CQR Routing Results

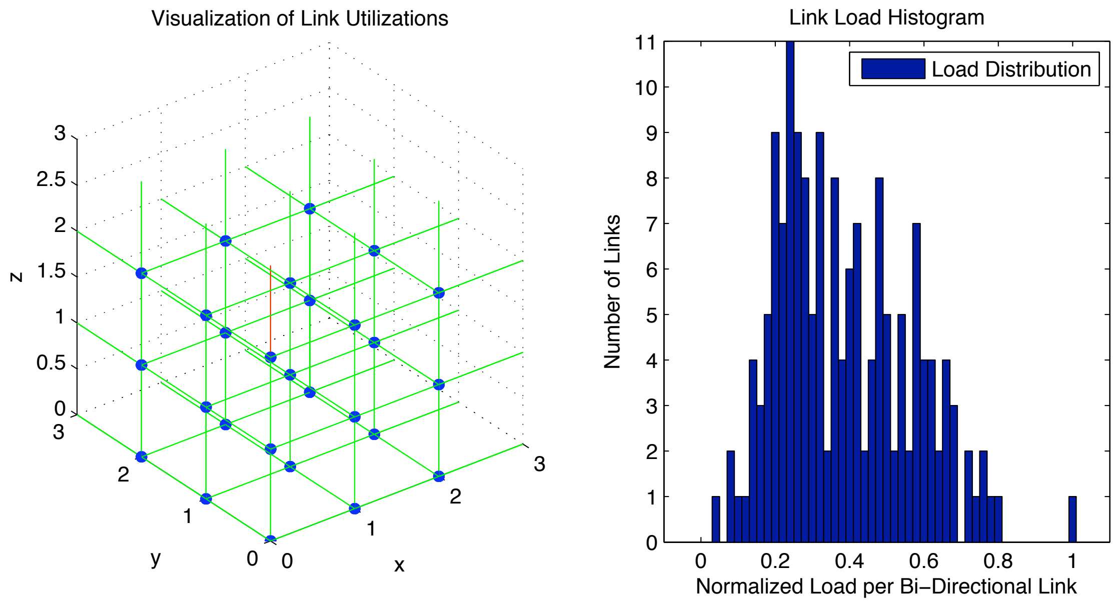


Figure D.27: *Results of CQR Routing using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

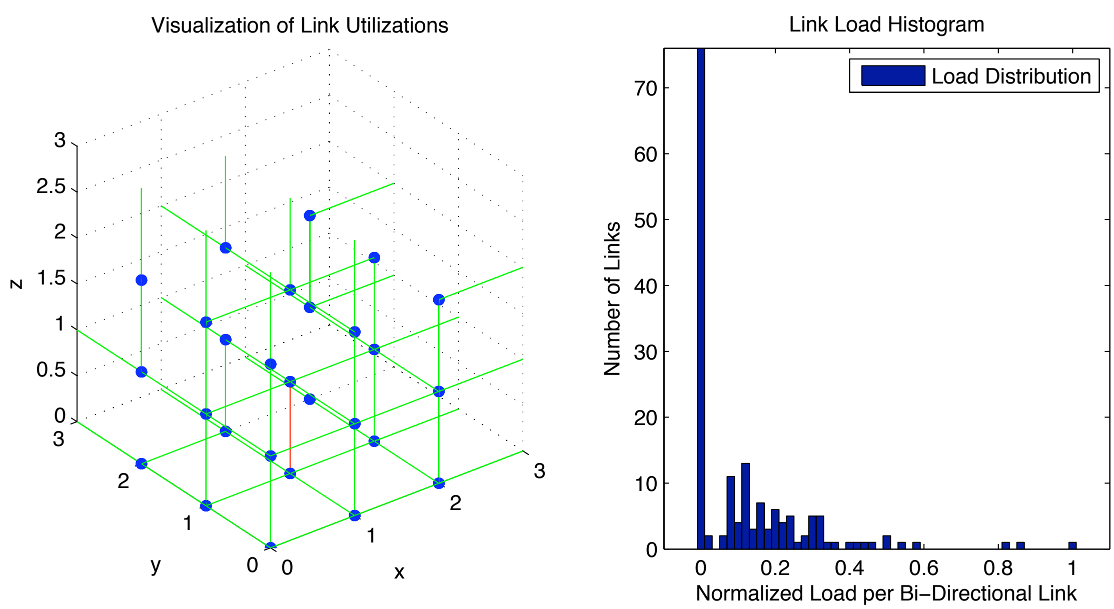


Figure D.28: *Results of CQR Routing using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

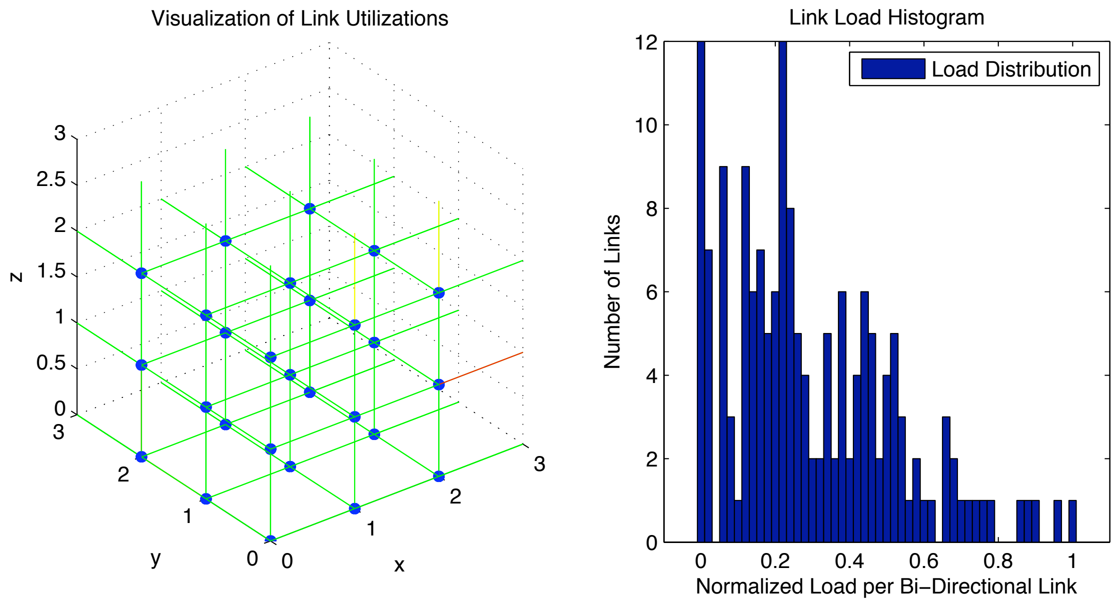


Figure D.29: *Results of CQR Routing using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

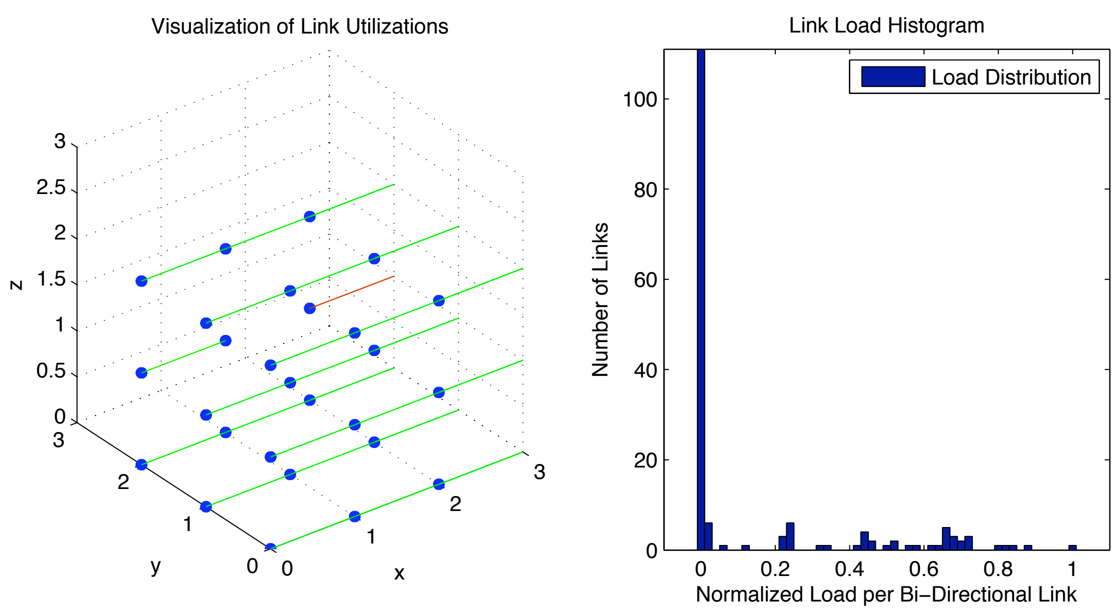


Figure D.30: *Results of CQR Routing using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

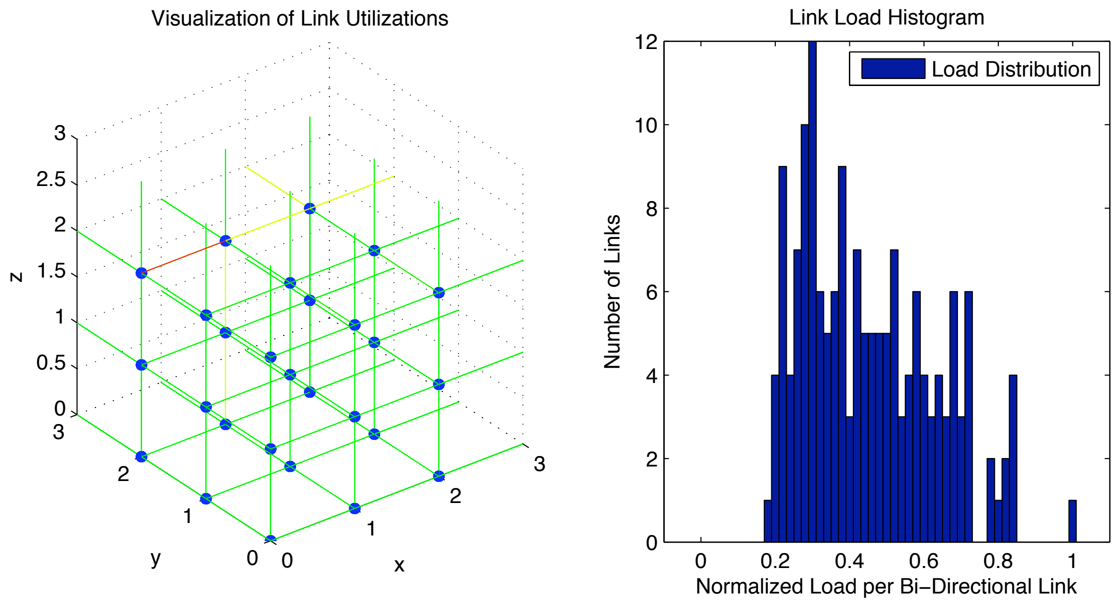


Figure D.31: *Results of CQR Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

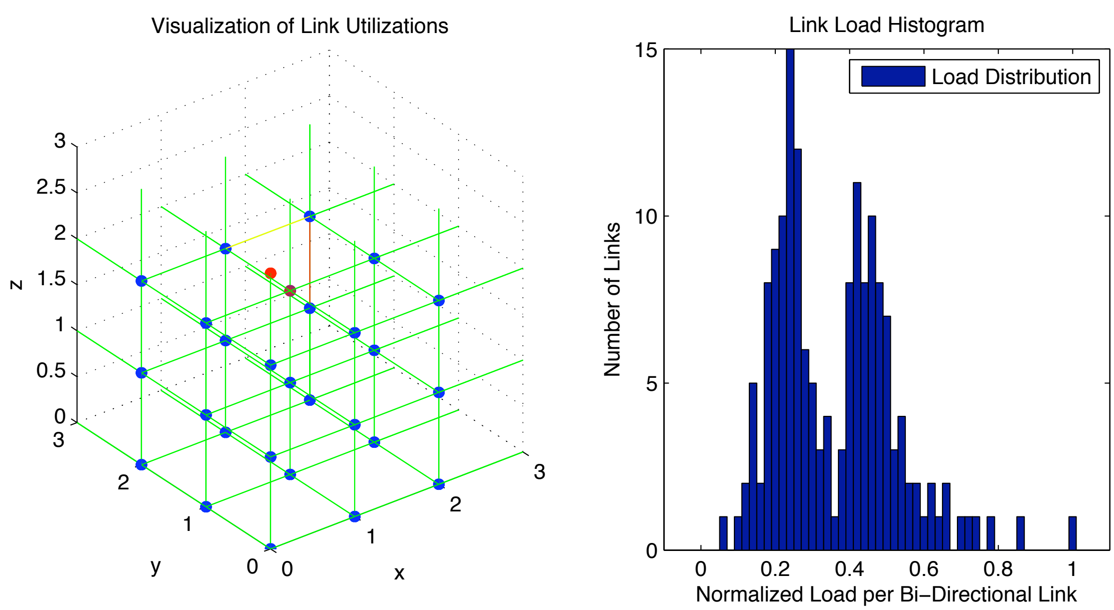


Figure D.32: *Results of CQR Routing using a flood traffic pattern (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

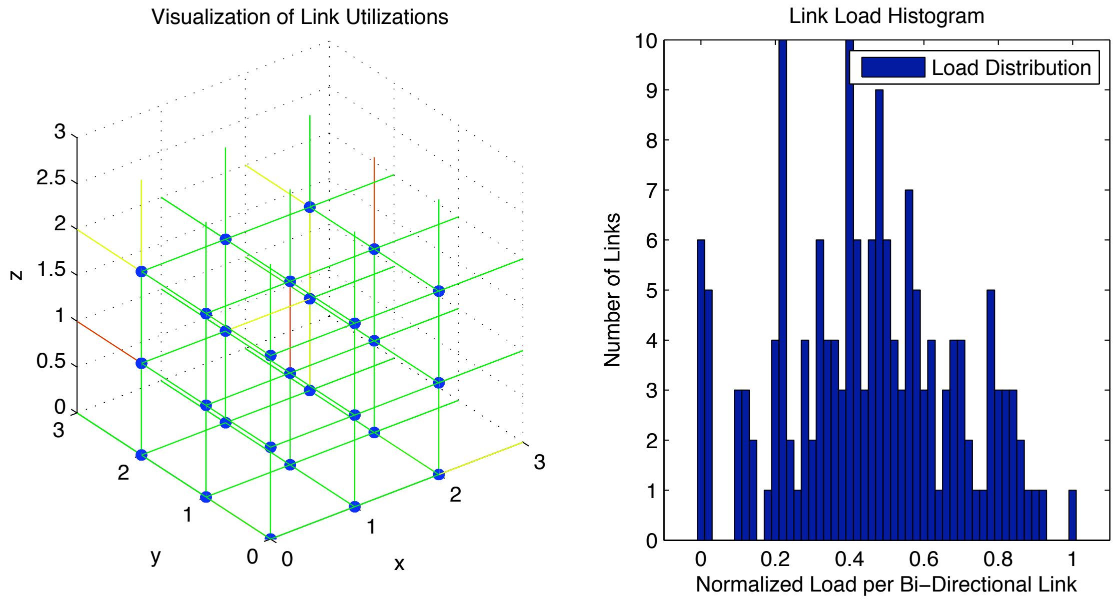


Figure D.33: *Results of Enhanced CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

D.4.3 Enhanced CQR Routing Results

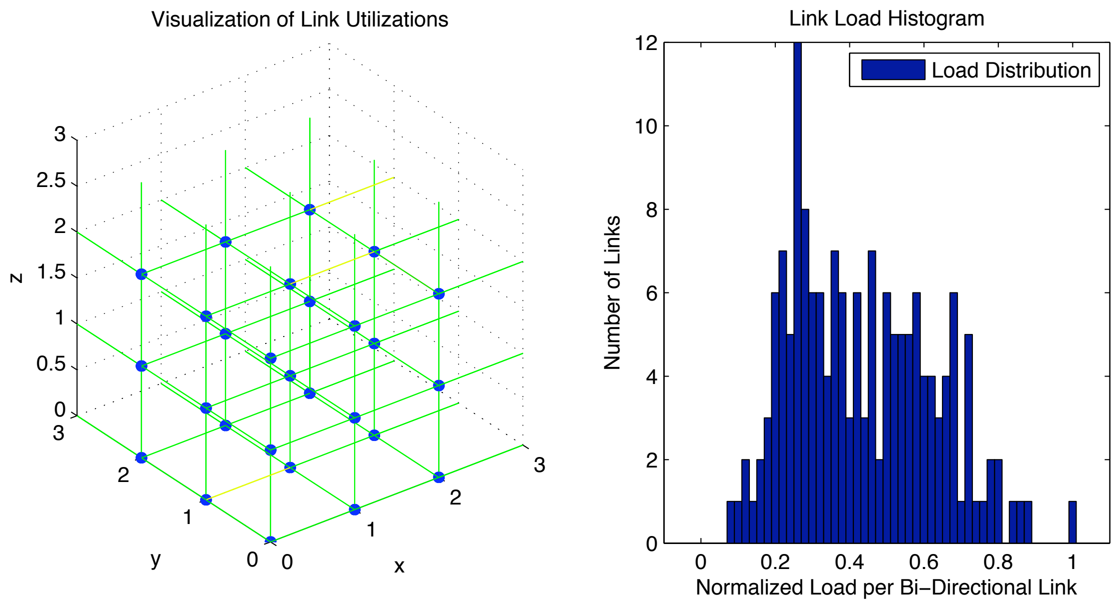


Figure D.34: *Results of Enhanced CQR Routing using a uniform random traffic pattern (UR). For each node, a value of 0-9 nodes was assigned as having 0-9 individual traffic demands.*

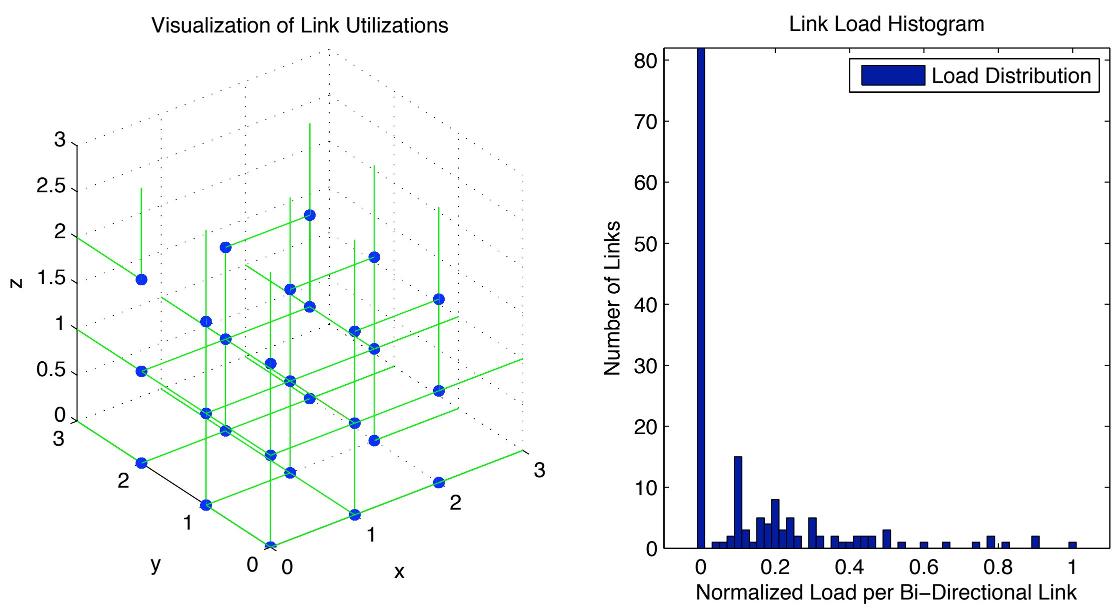


Figure D.35: *Results of Enhanced CQR Routing using a bit complement traffic pattern (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

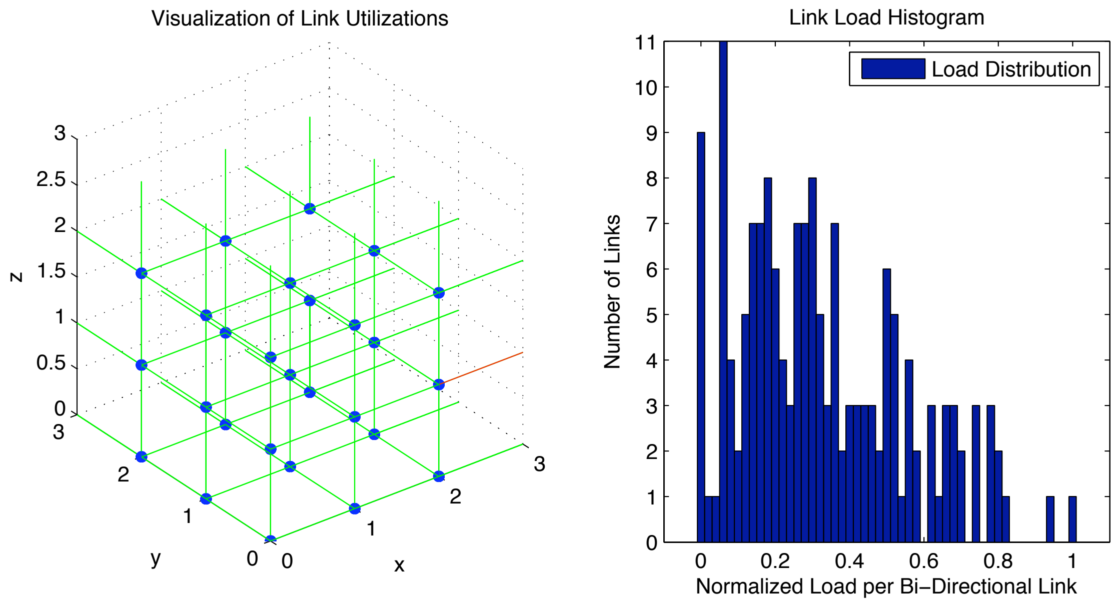


Figure D.36: *Results of Enhanced CQR Routing using a transpose traffic pattern (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

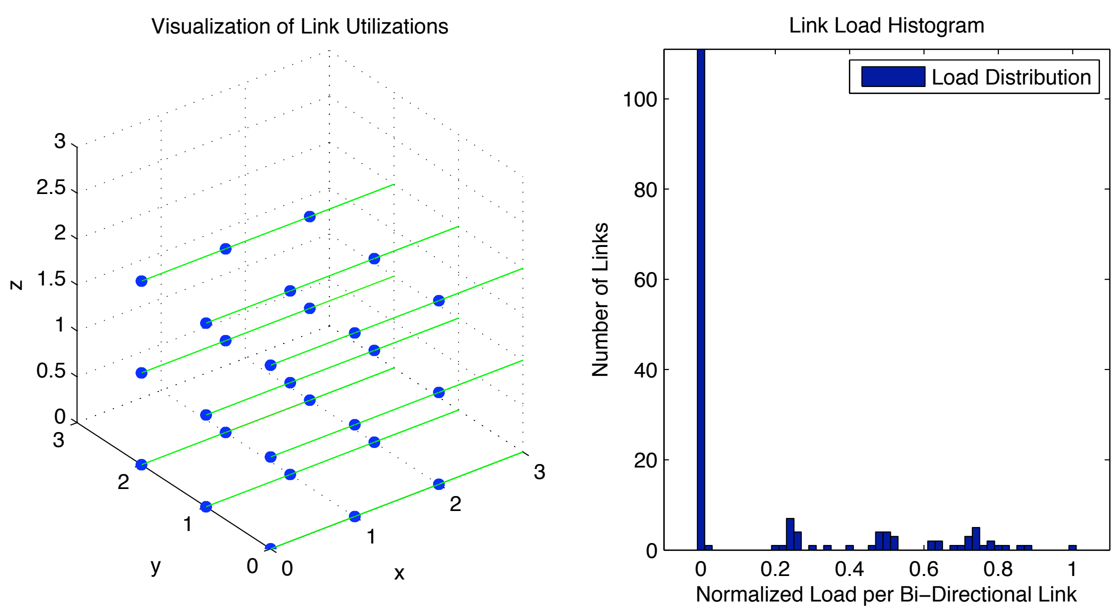


Figure D.37: *Results of Enhanced CQR Routing using a tornado traffic pattern (TOR). Three individual traffic demands were assigned according to Table 2.1.*

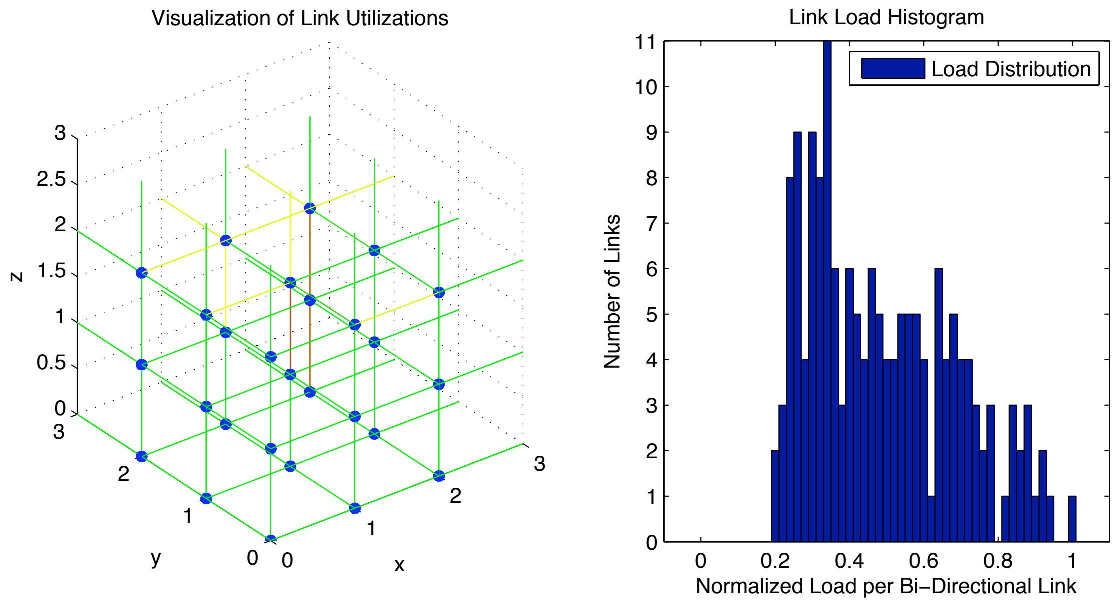


Figure D.38: *Results of Enhanced CQR Routing using a flood traffic pattern (FL). Two individual traffic demands were assigned to each node from each node.*

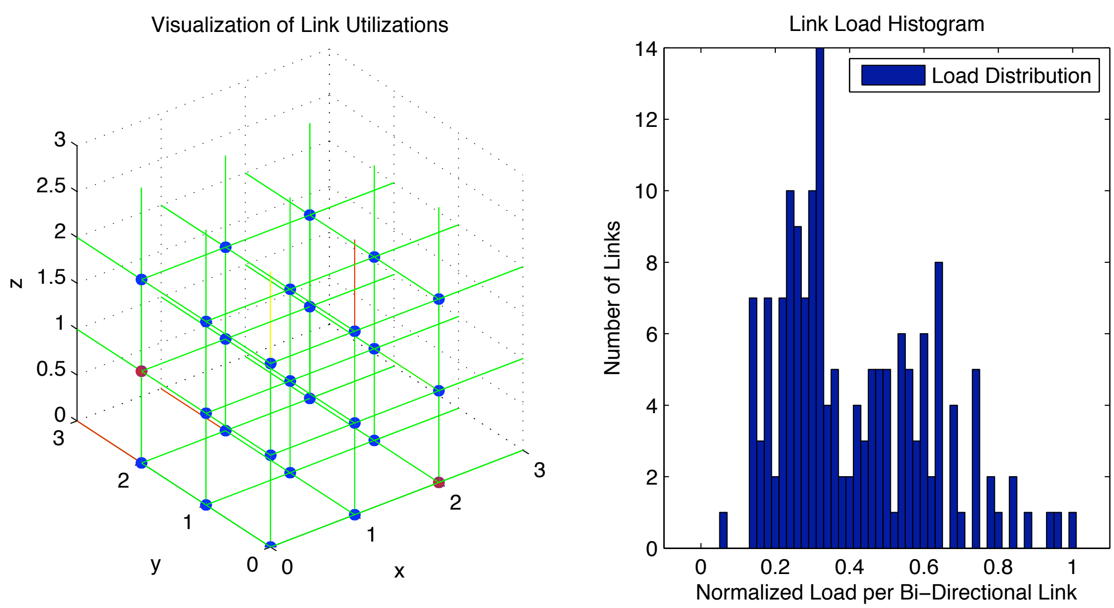


Figure D.39: *Results of Enhanced CQR Routing using a flood traffic pattern (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

Appendix E

Appendix E: Full Matlab Simulation Results

E.1 Results from the 3-ary 3-cube Topology

E.1.1 Static Algorithms

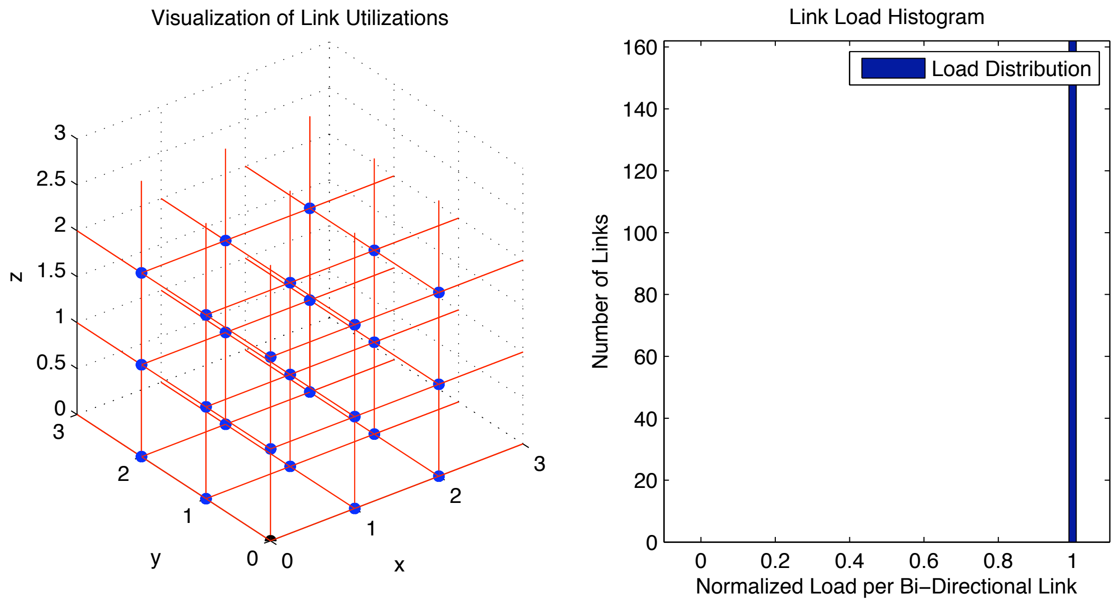


Figure E.1: *Results of Dimension Ordered Routing (DOR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

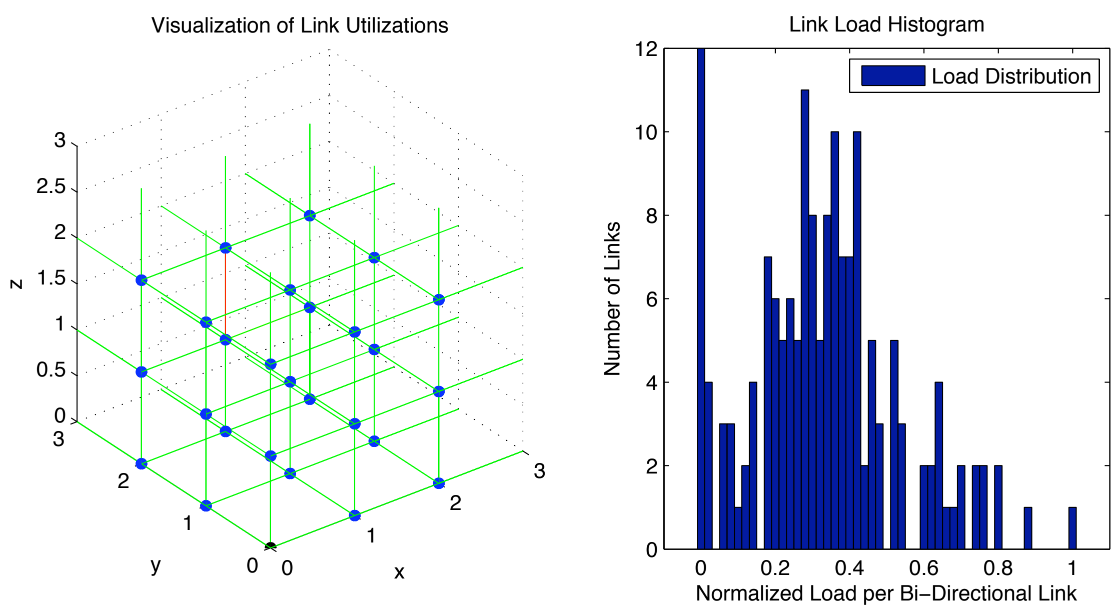


Figure E.2: *Results of Dimension Ordered Routing (DOR) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

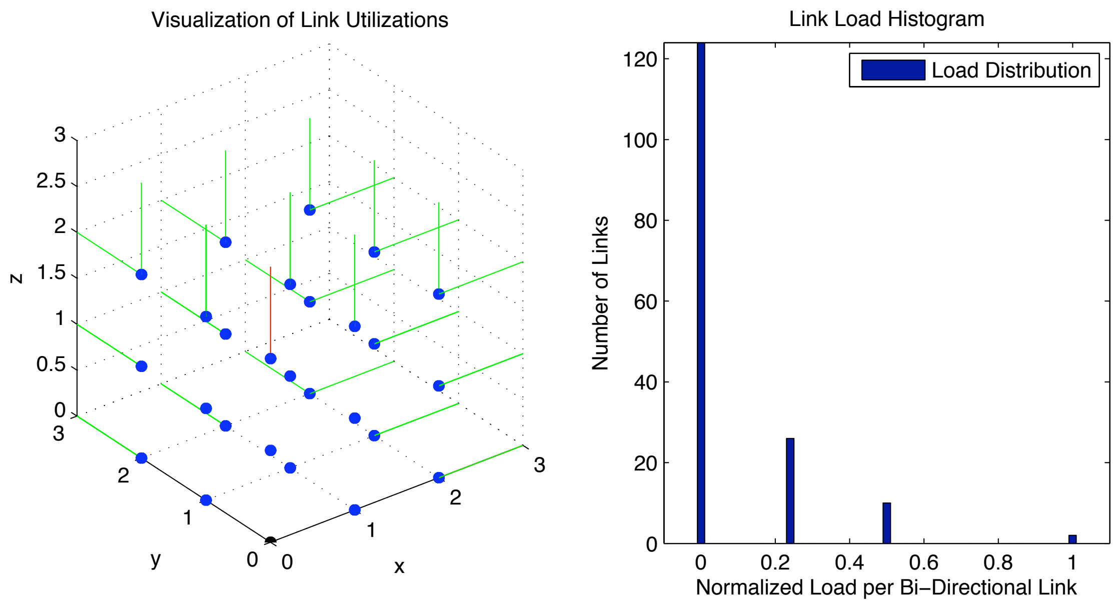


Figure E.3: *Results of Dimension Ordered Routing (DOR) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

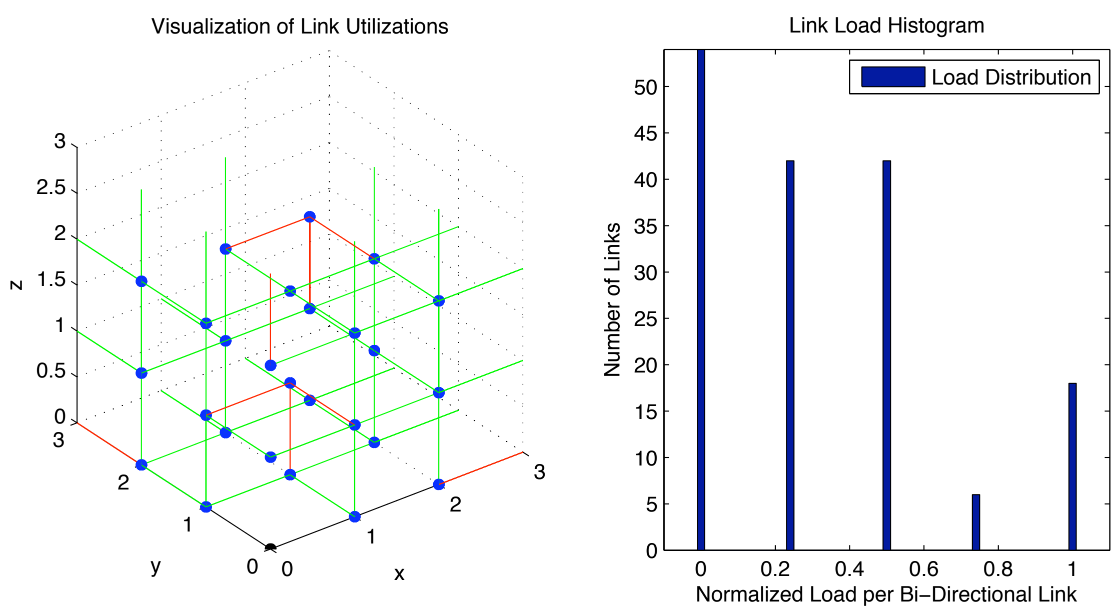


Figure E.4: *Results of Dimension Ordered Routing (DOR) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all* x,y,z *values.*

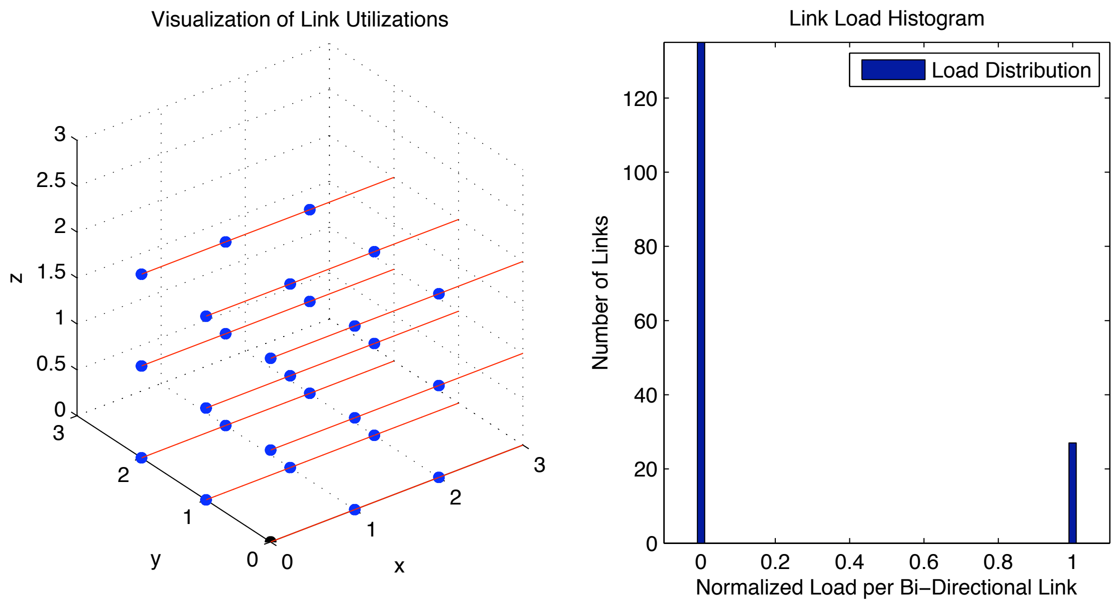


Figure E.5: *Results of Dimension Ordered Routing (DOR) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

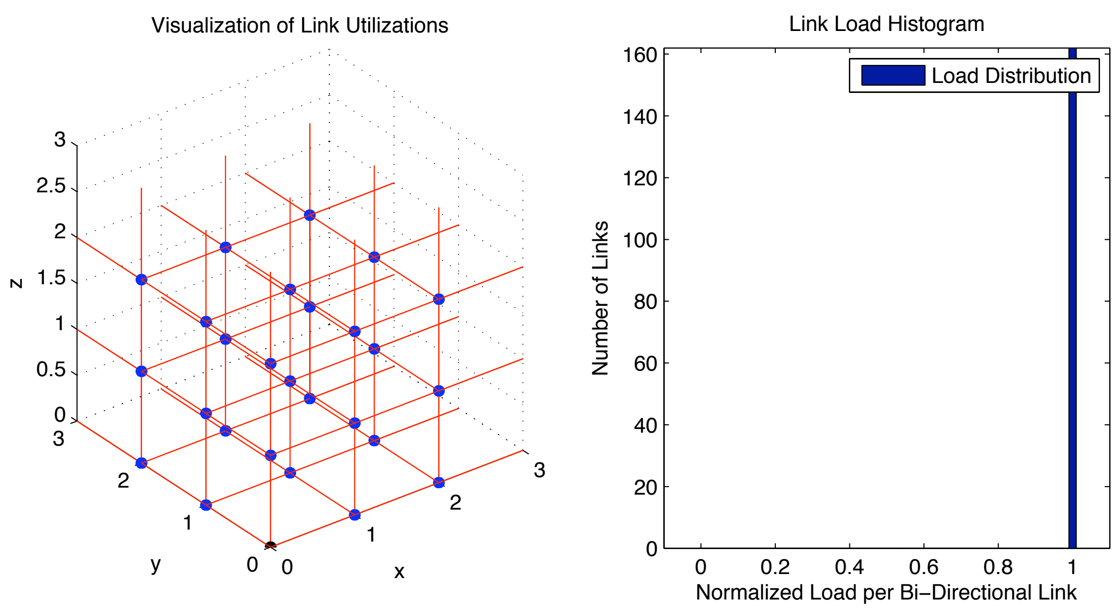


Figure E.6: *Results of Dimension Ordered Routing (DOR) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

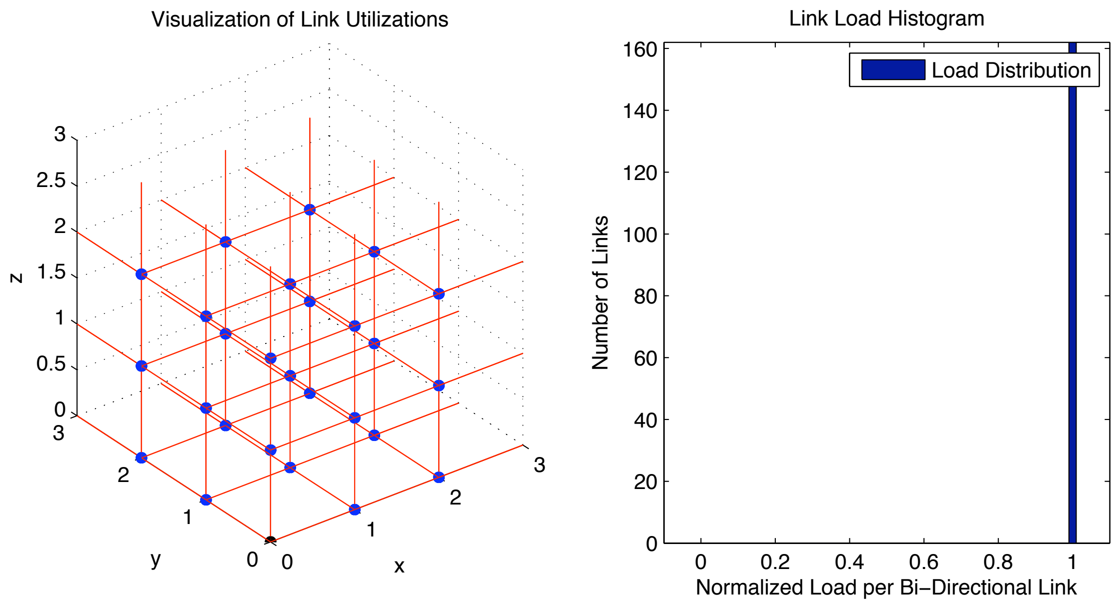


Figure E.7: *Results of Direction Ordered Routing (DIR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

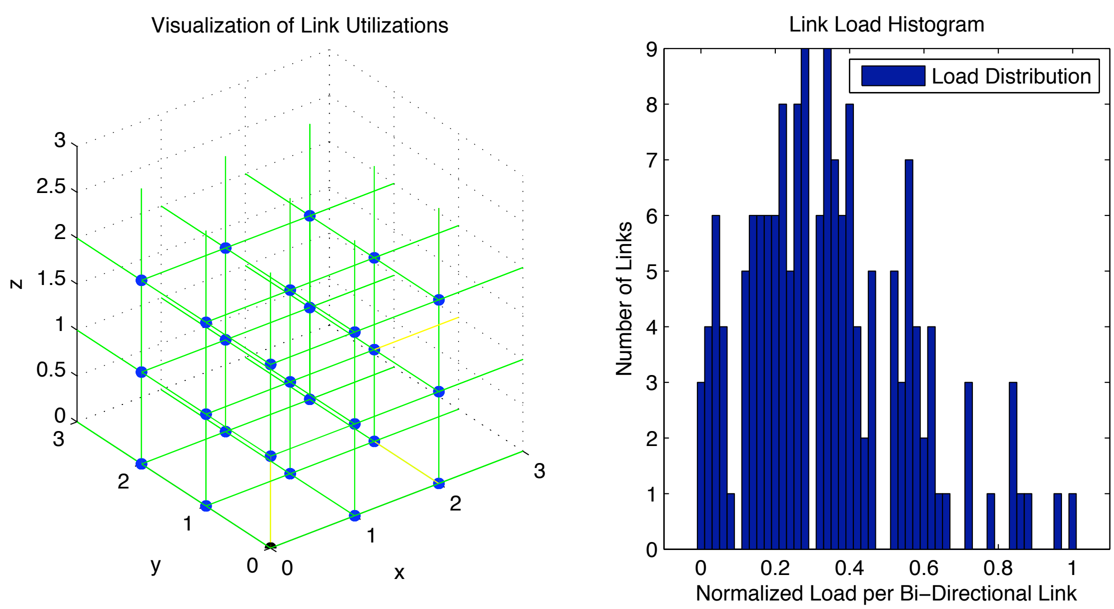


Figure E.8: *Results of Direction Ordered Routing (DIR) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

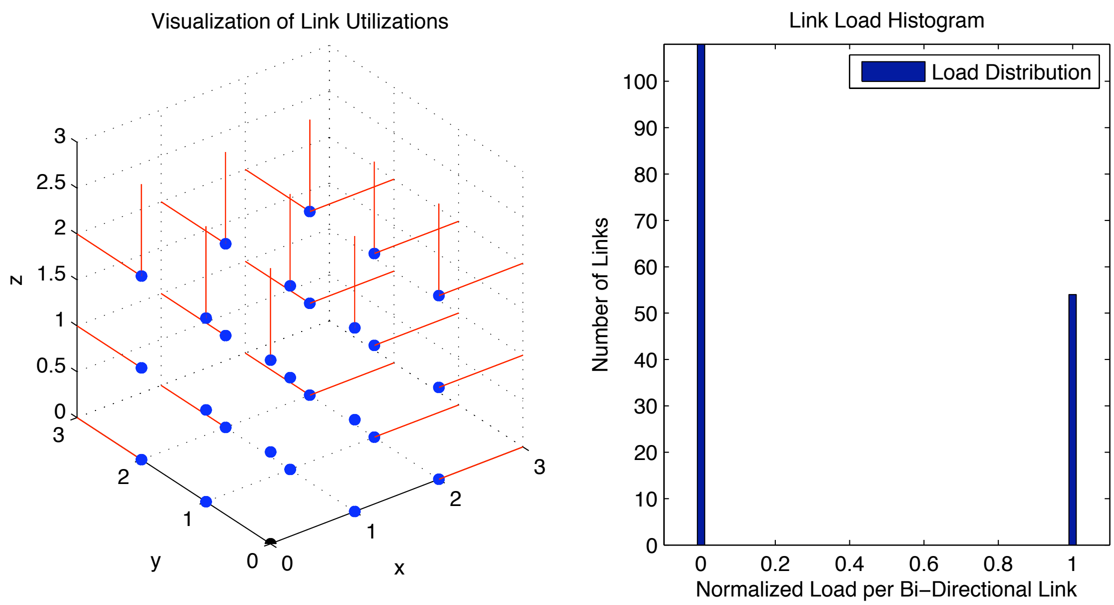


Figure E.9: *Results of Direction Ordered Routing (DIR) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

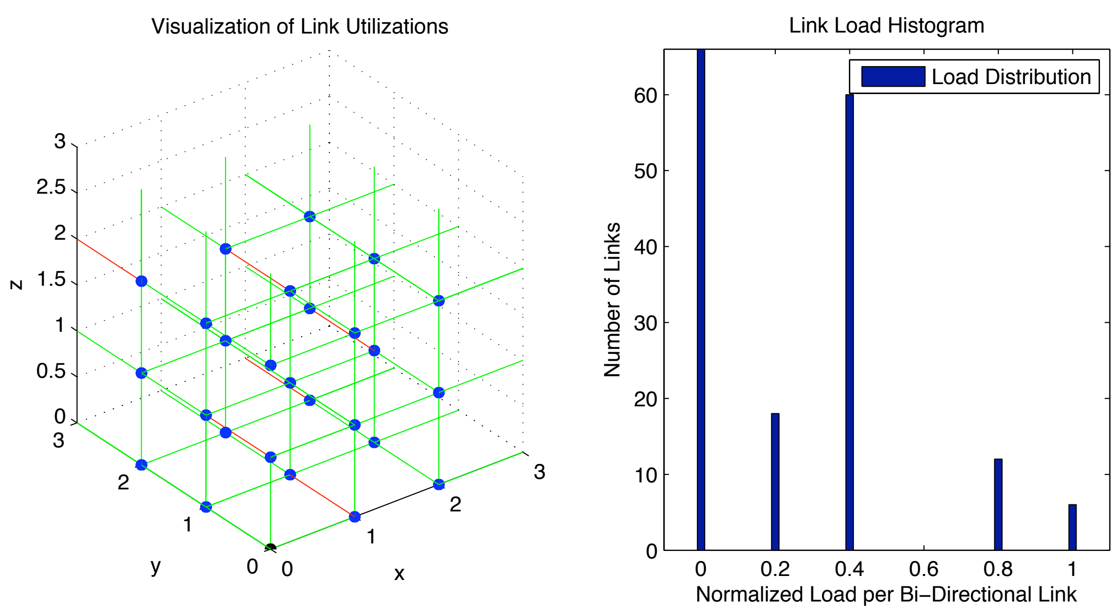


Figure E.10: *Results of Direction Ordered Routing (DIR) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

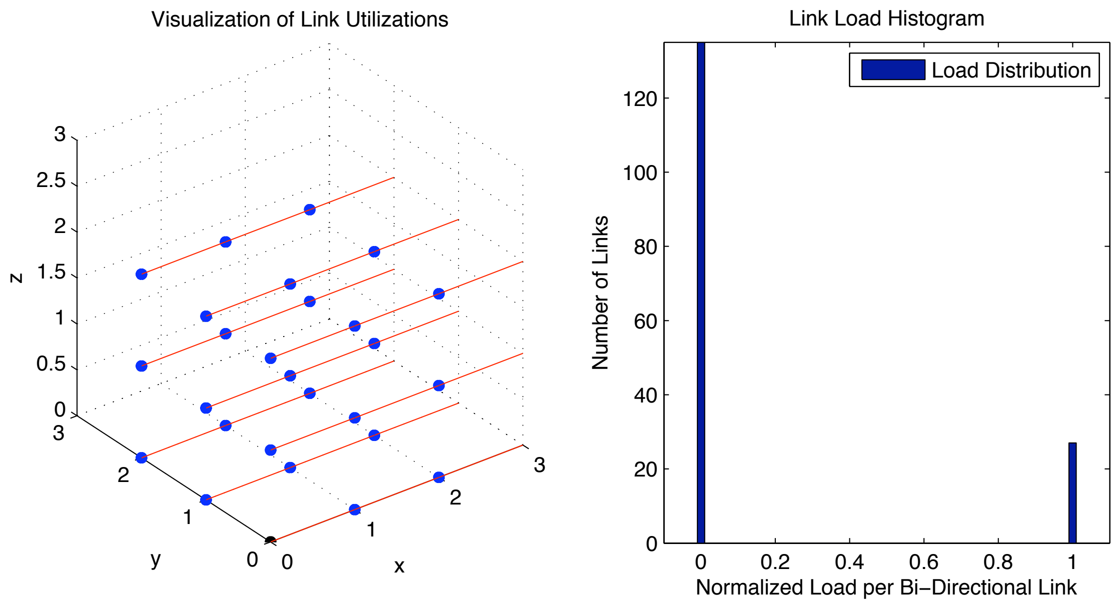


Figure E.11: *Results of Direction Ordered Routing (DIR) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

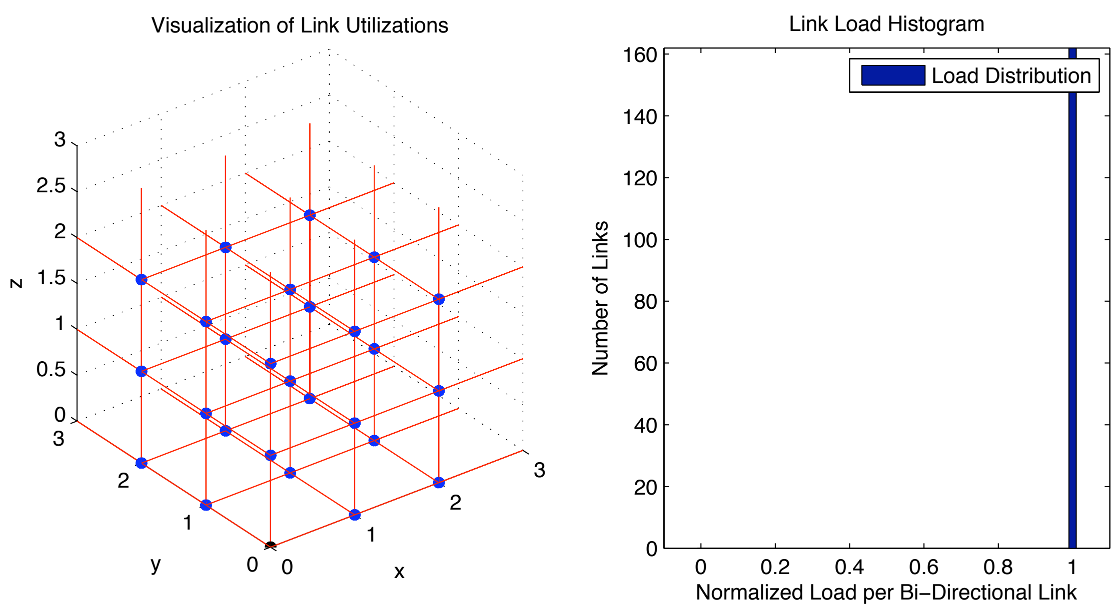


Figure E.12: *Results of Direction Ordered Routing (DIR) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

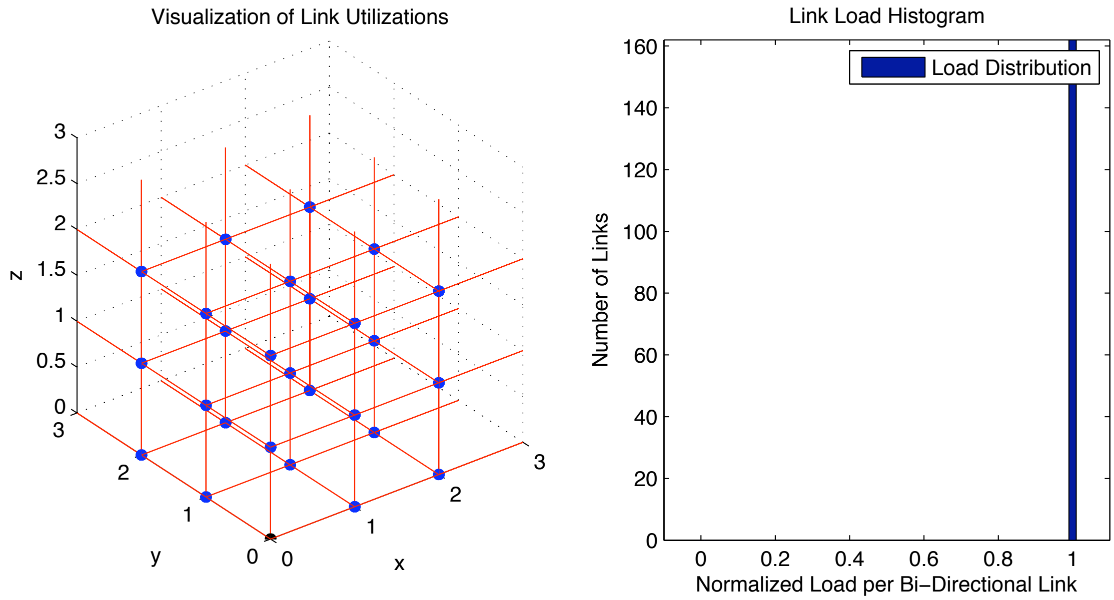


Figure E.13: *Results of Minimal Oblivious Routing (MO) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

E.1.2 Oblivious Algorithms

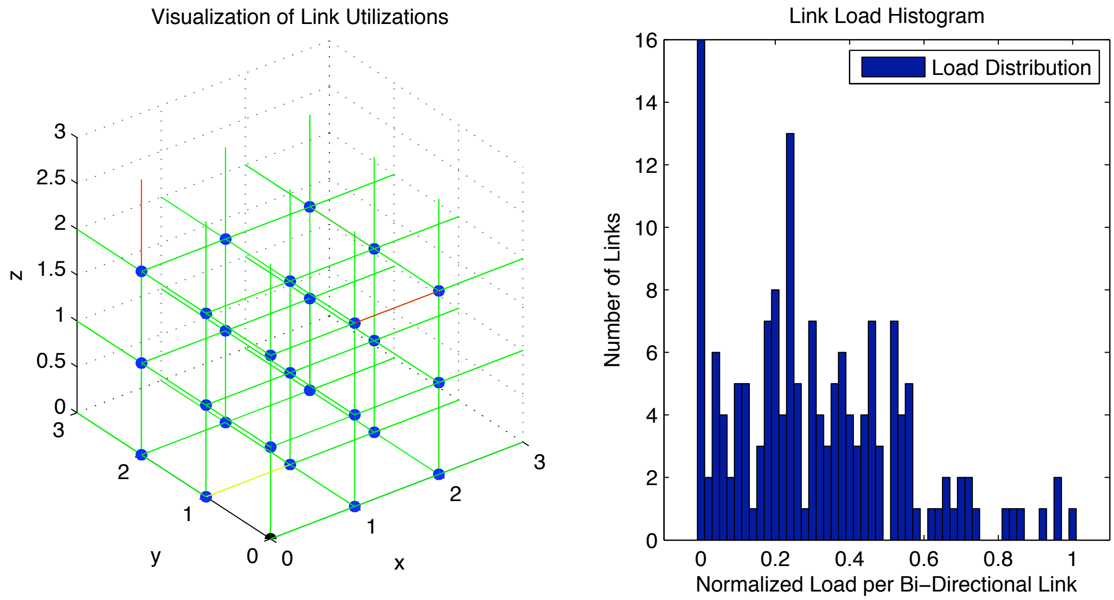


Figure E.14: *Results of Minimal Oblivious Routing (MO) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

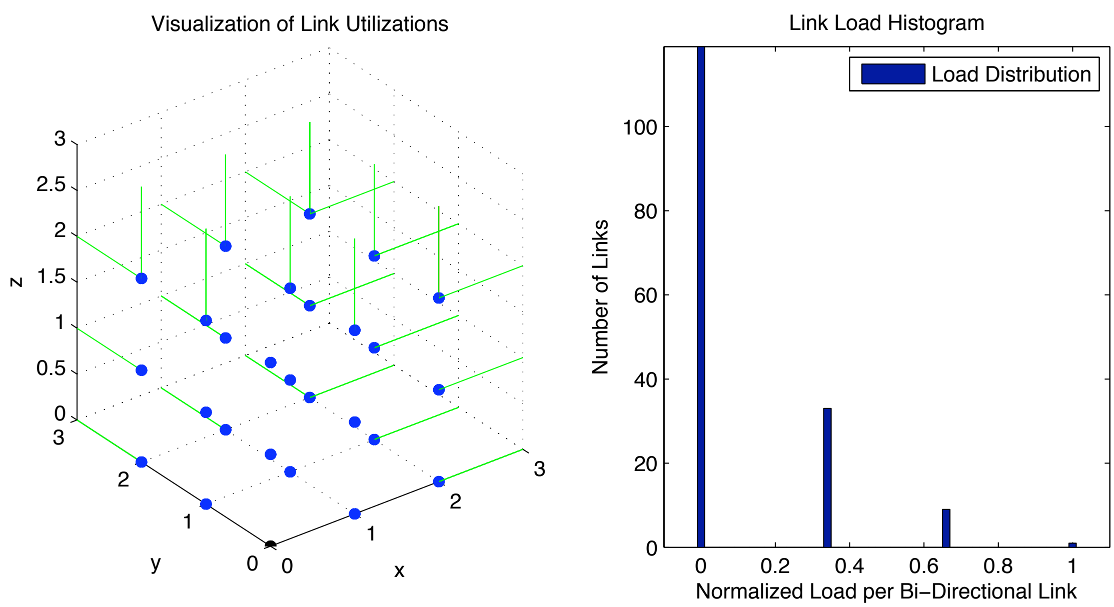


Figure E.15: *Results of Minimal Oblivious Routing (MO) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

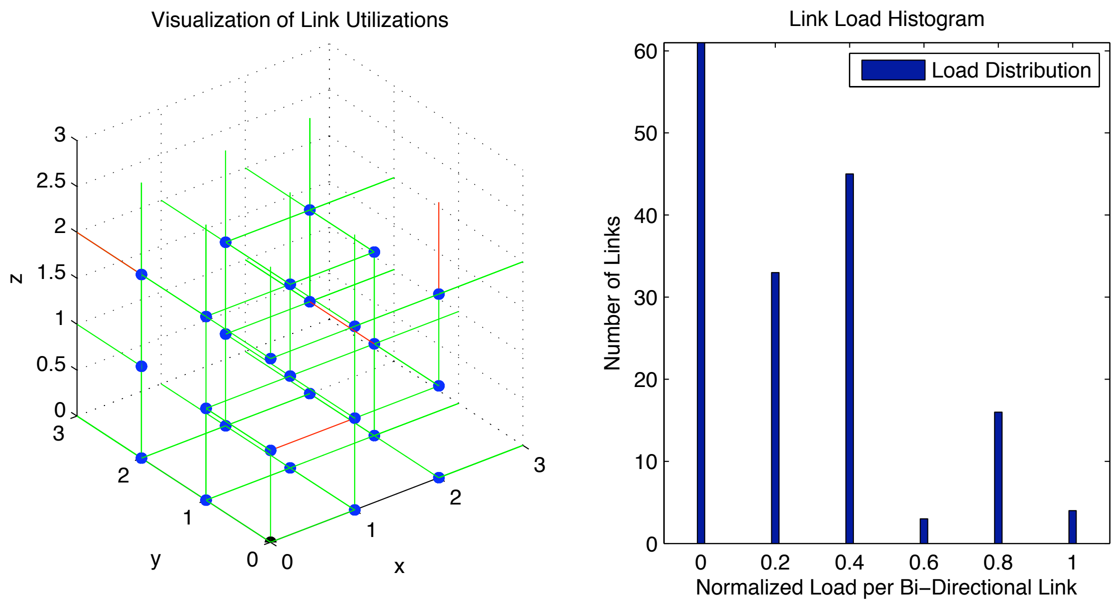


Figure E.16: *Results of Minimal Oblivious Routing (MO) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

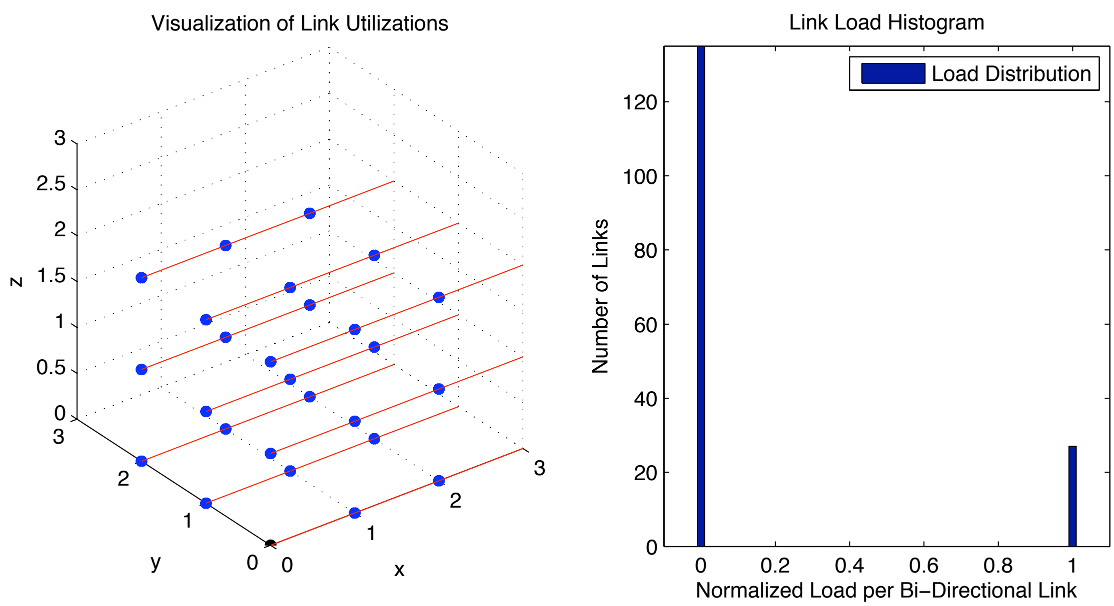


Figure E.17: *Results of Minimal Oblivious Routing (MO) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

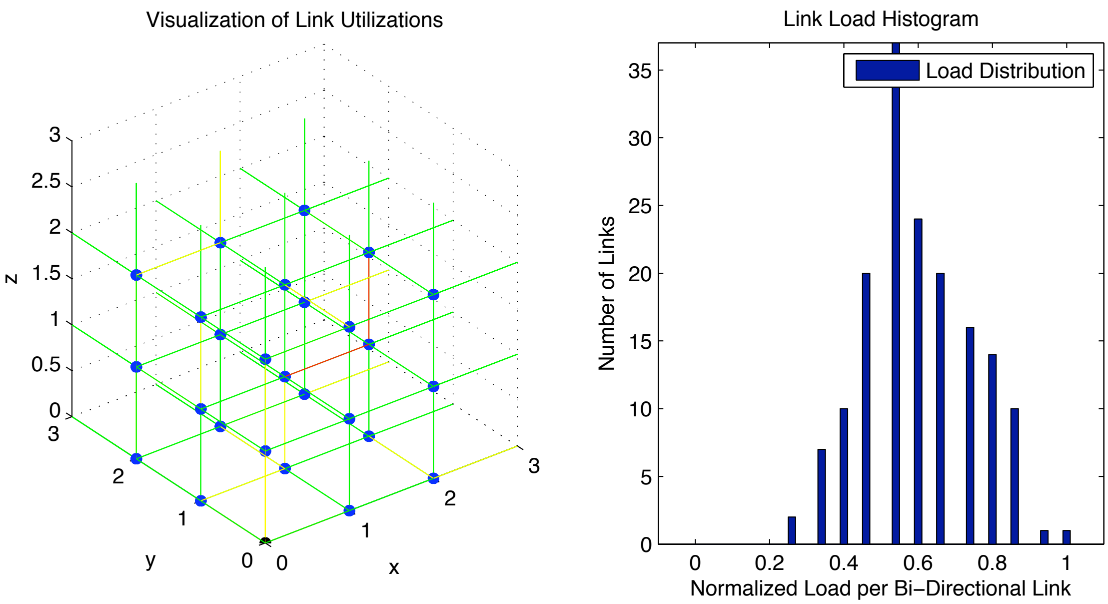


Figure E.18: *Results of Minimal Oblivious Routing (MO) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

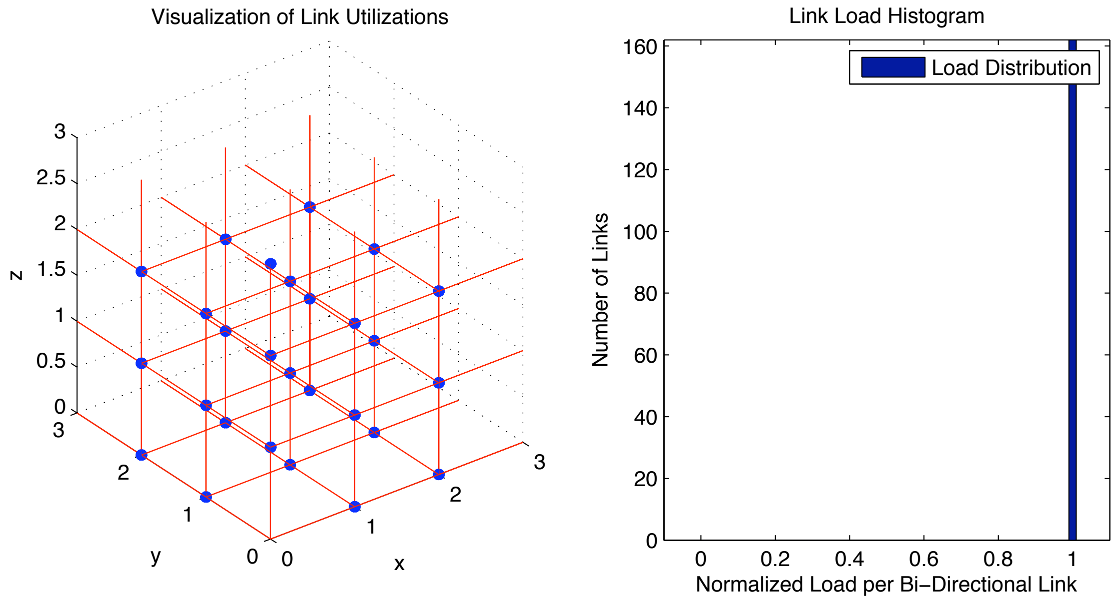


Figure E.19: *Results of Minimal Adaptive Routing (MA) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

E.1.3 Adaptive Algorithms

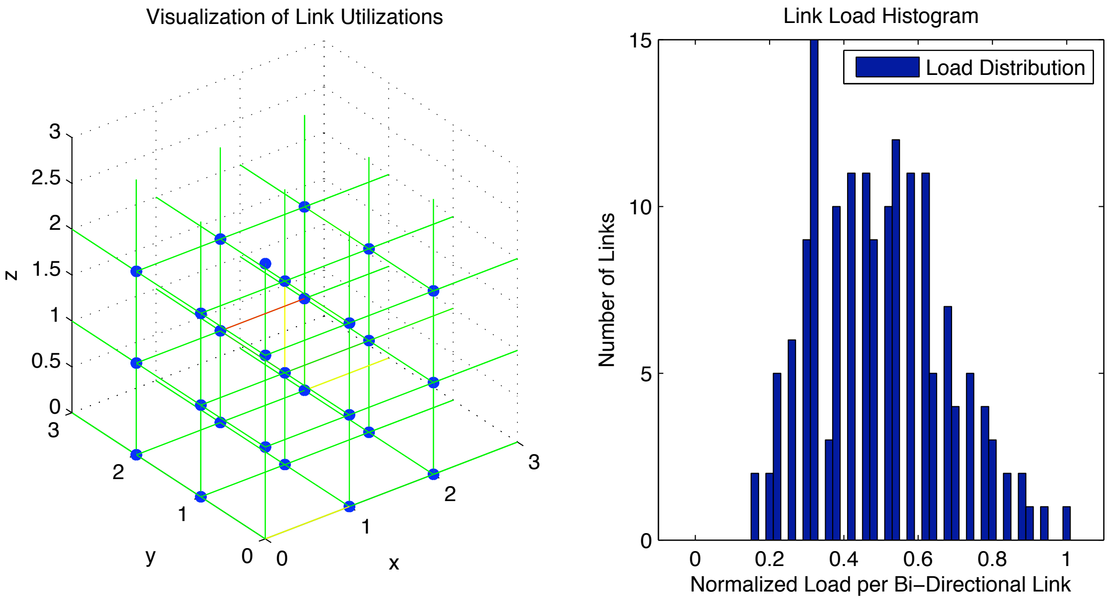


Figure E.20: *Results of Minimal Adaptive Routing (MA) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

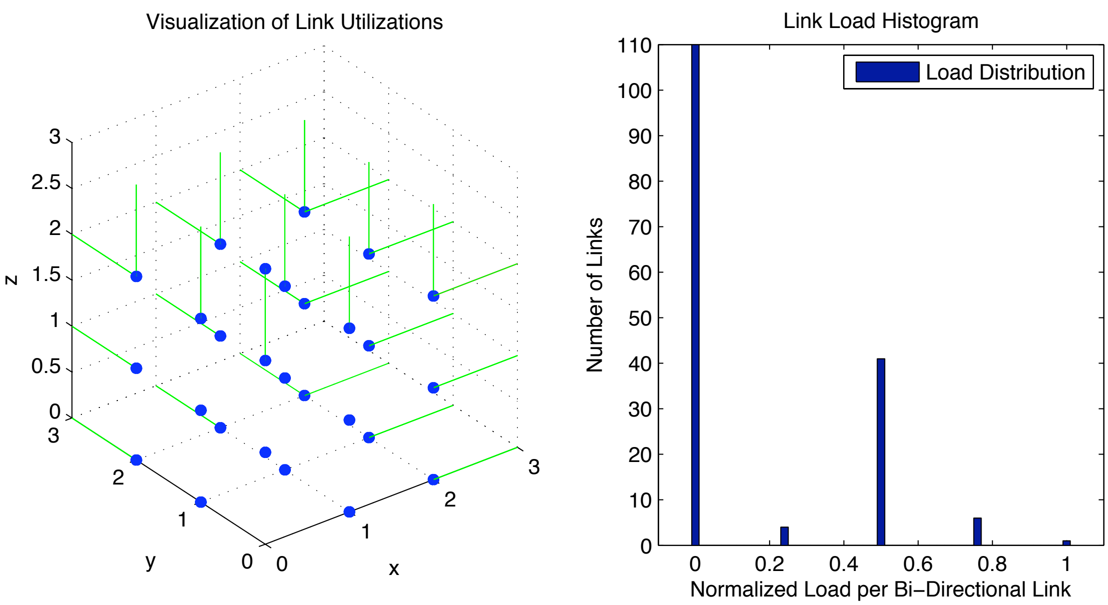


Figure E.21: *Results of Minimal Adaptive Routing (MA) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

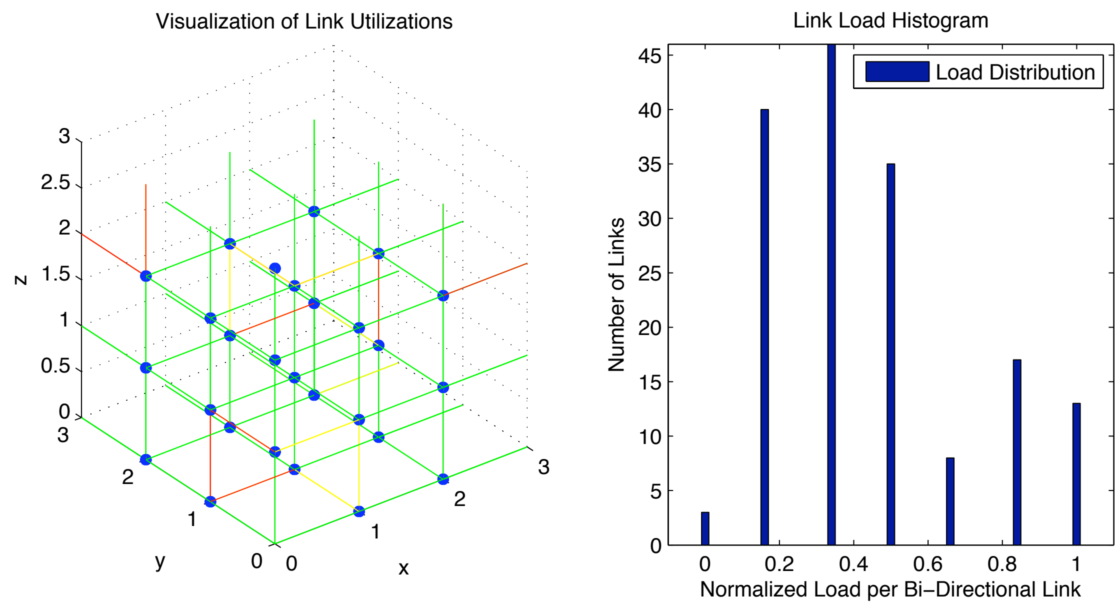


Figure E.22: *Results of Minimal Adaptive Routing (MA) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

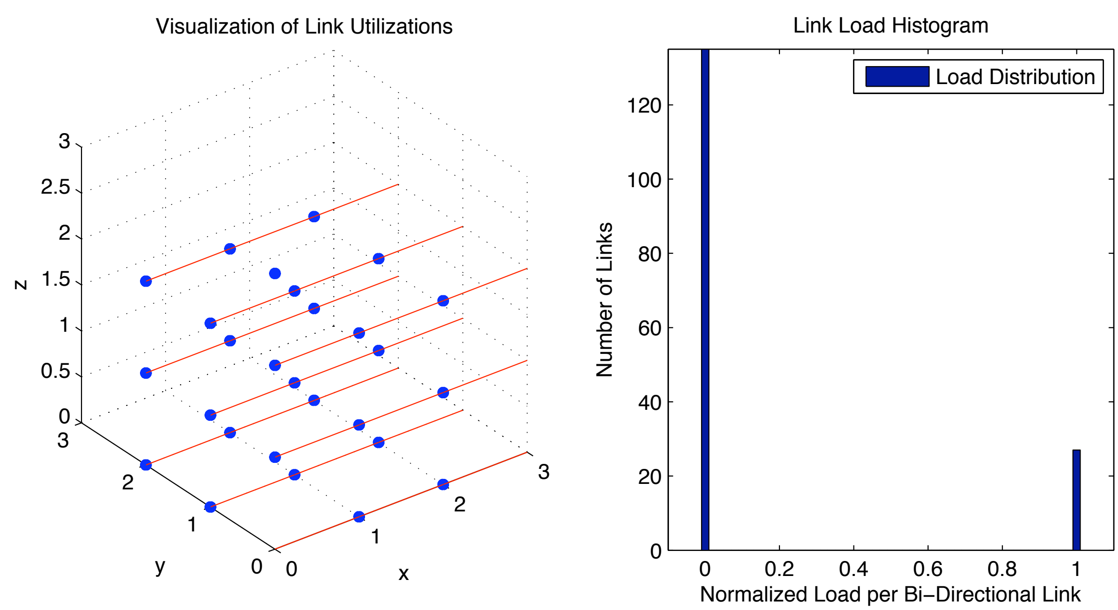


Figure E.23: *Results of Minimal Adaptive Routing (MA) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

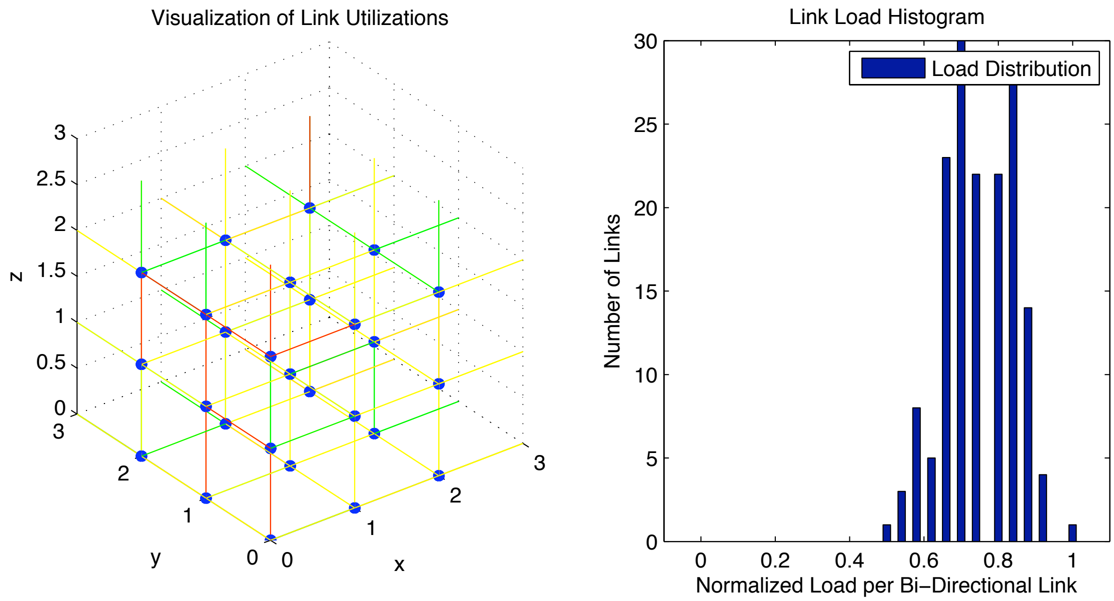


Figure E.24: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

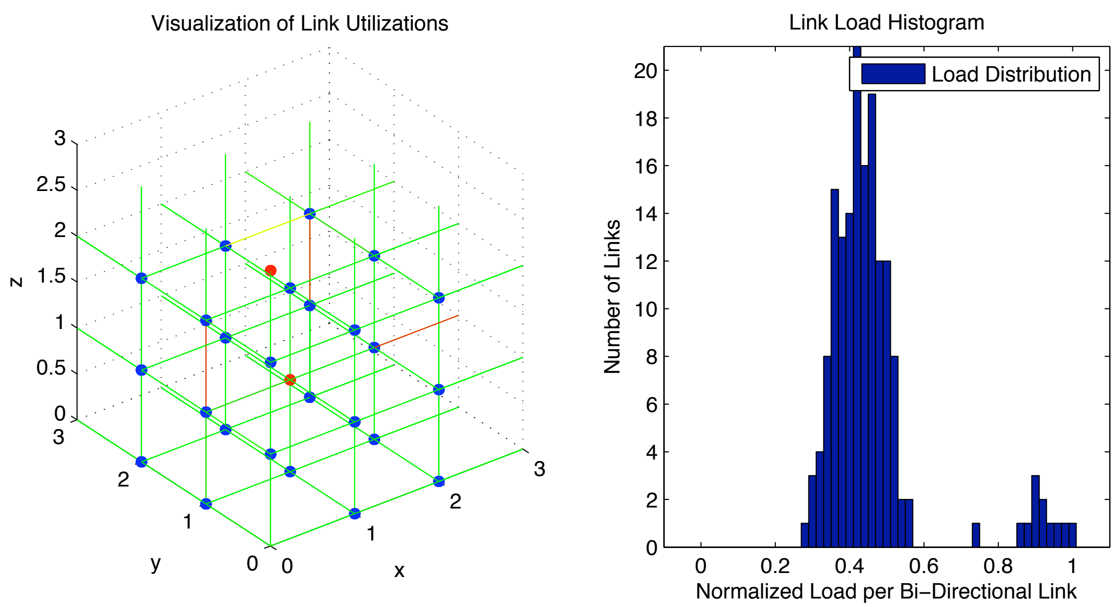


Figure E.25: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

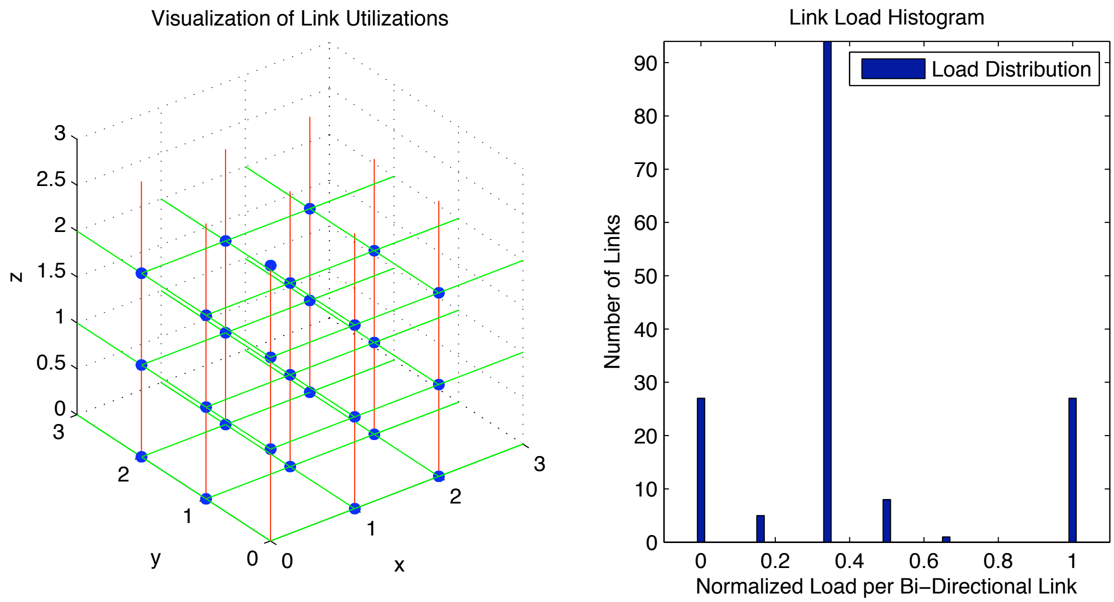


Figure E.26: *Results of CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

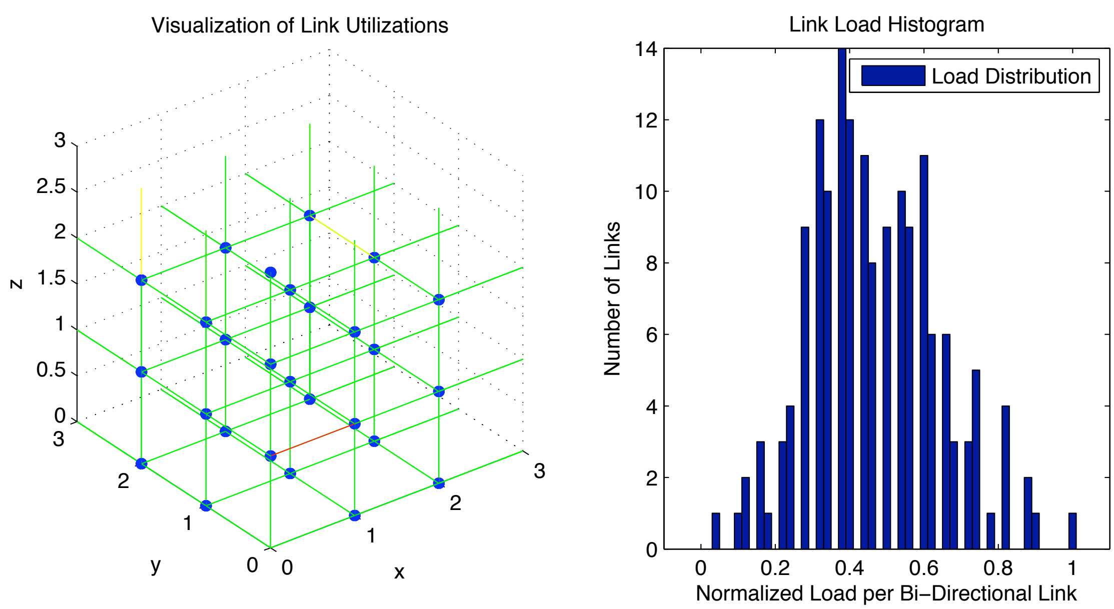


Figure E.27: *Results of CQR Routing using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

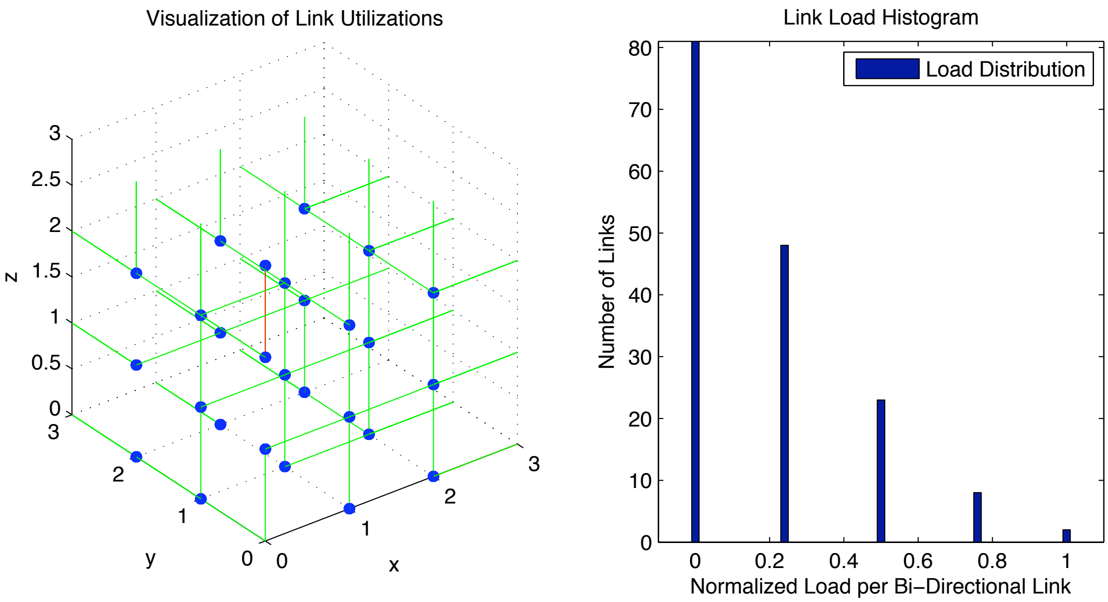


Figure E.28: *Results of CQR Routing using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

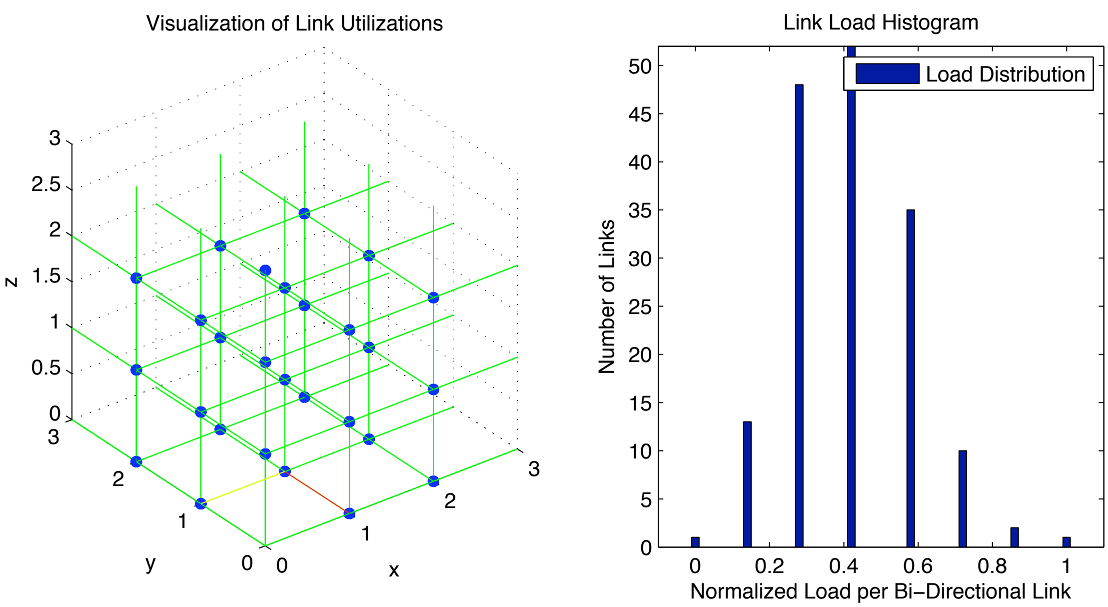


Figure E.29: *Results of CQR Routing using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

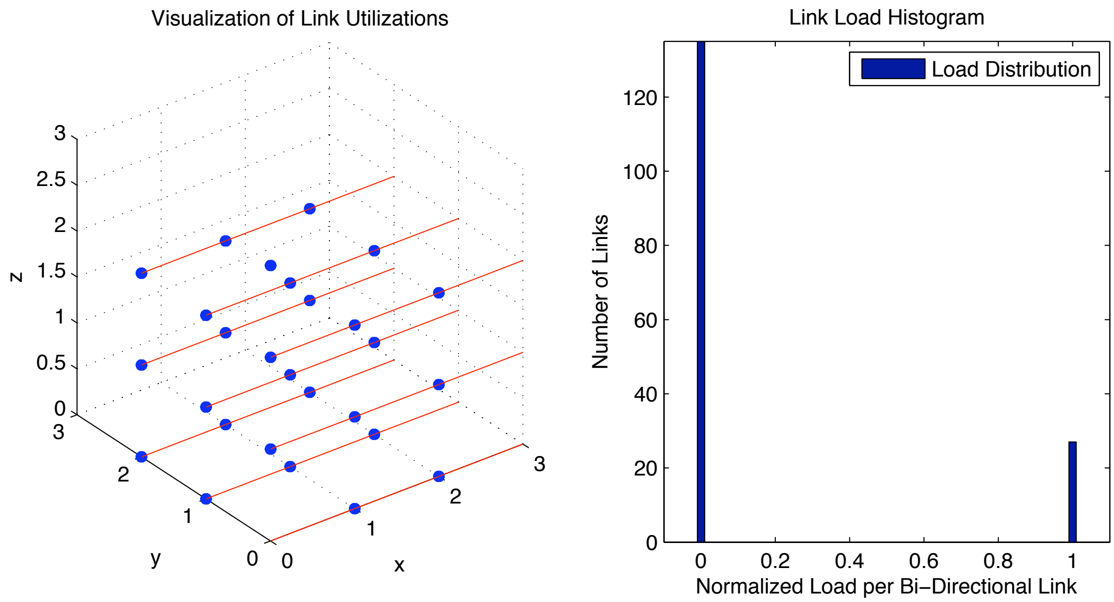


Figure E.30: *Results of CQR Routing using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

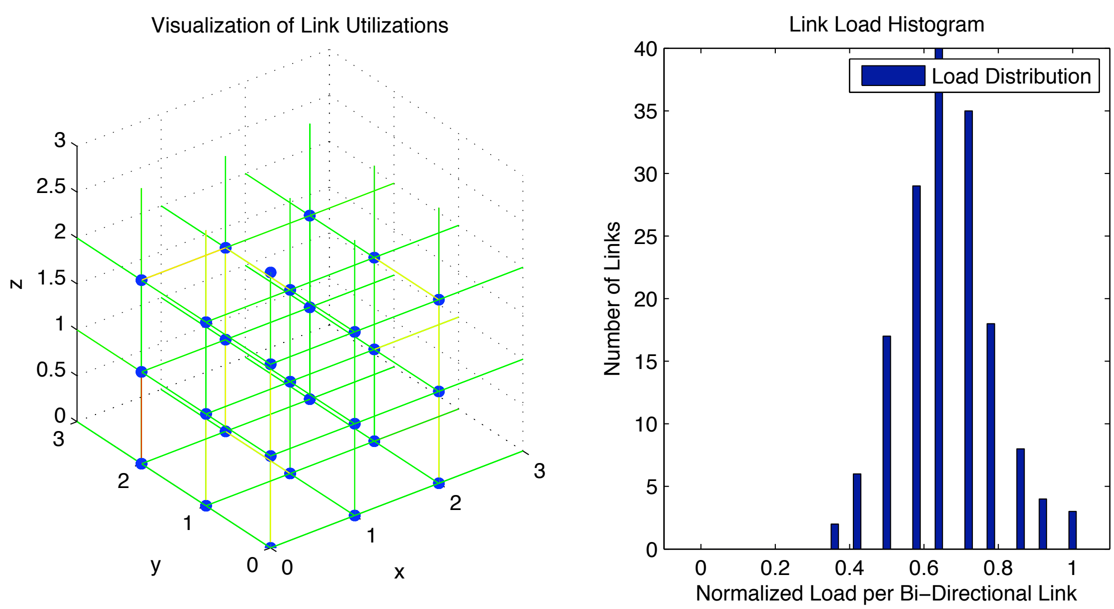


Figure E.31: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

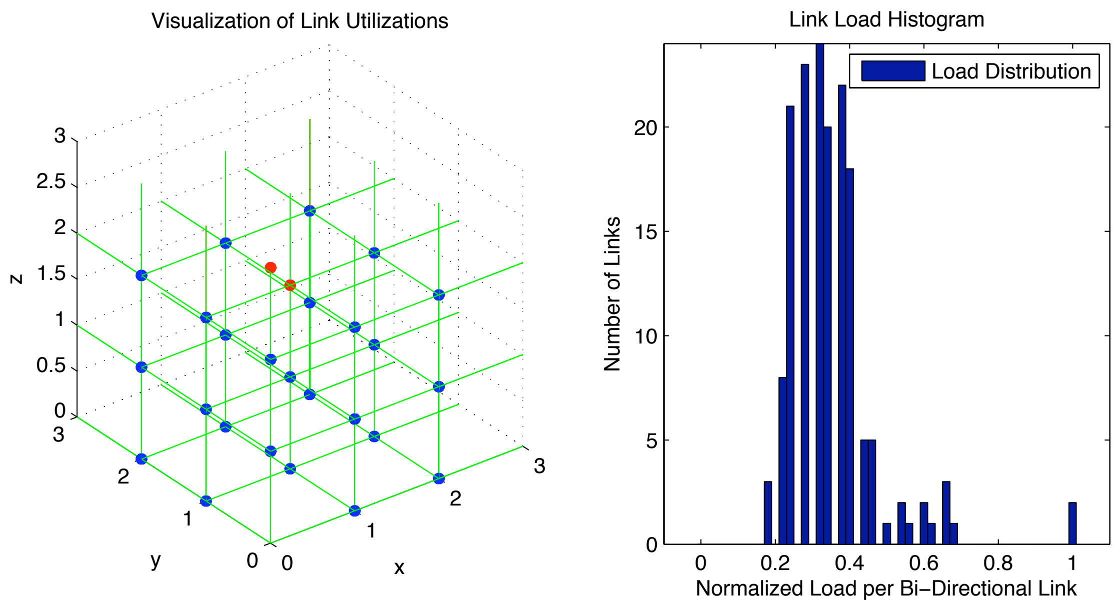


Figure E.32: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

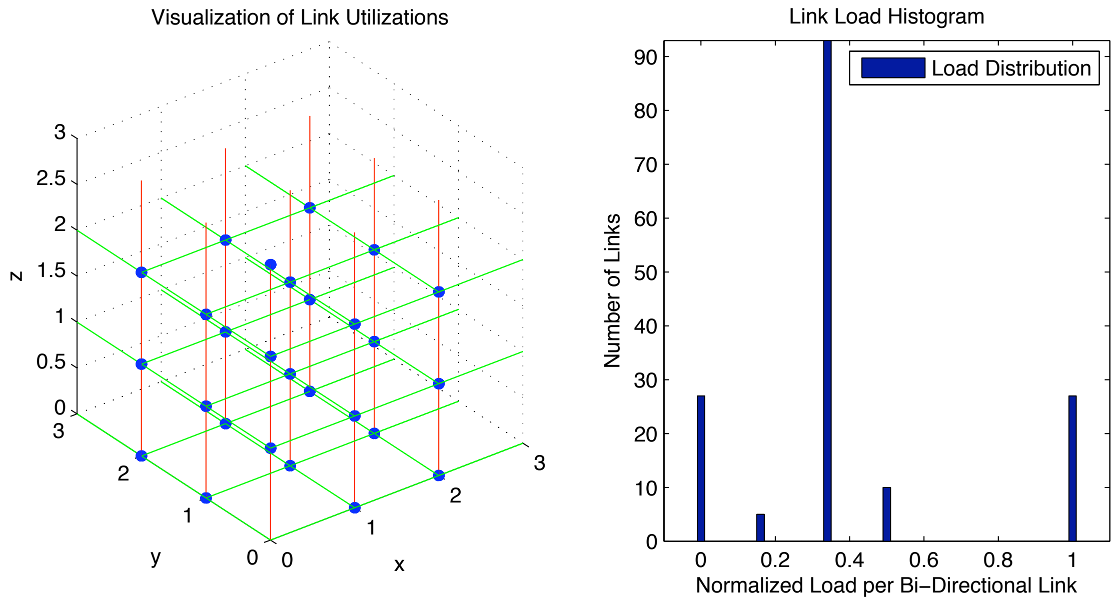


Figure E.33: *Results of Enhanced CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

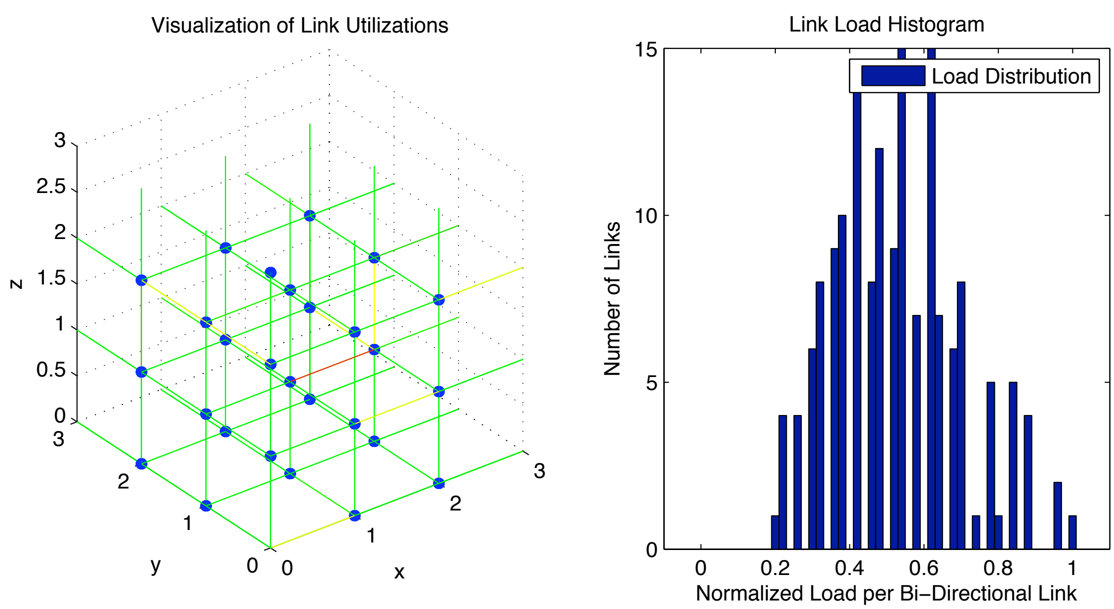


Figure E.34: *Results of Enhanced CQR Routing using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

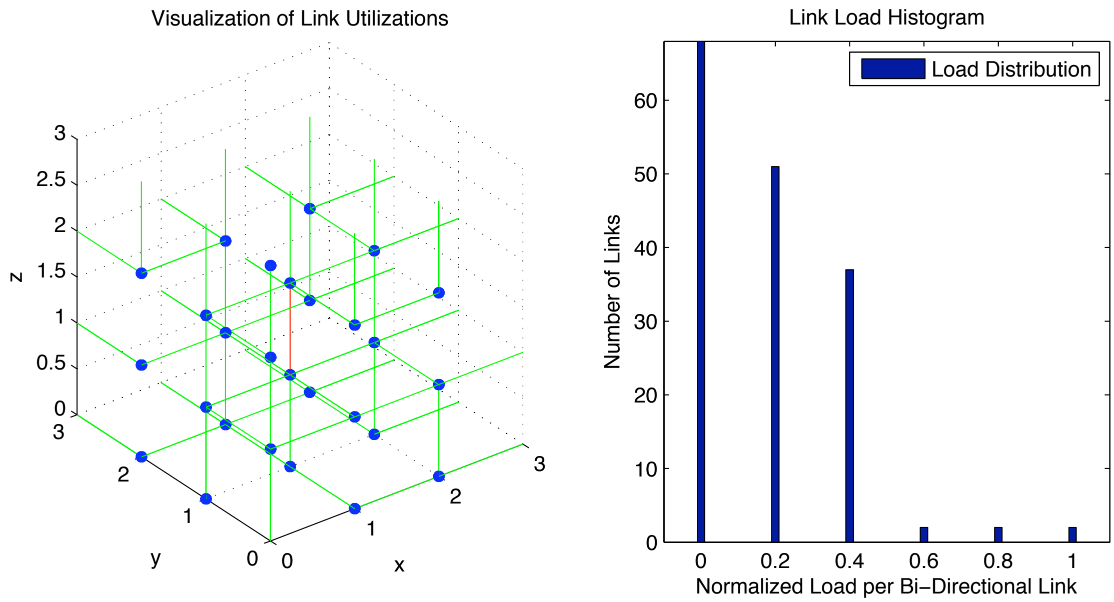


Figure E.35: *Results of CQR Routing using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

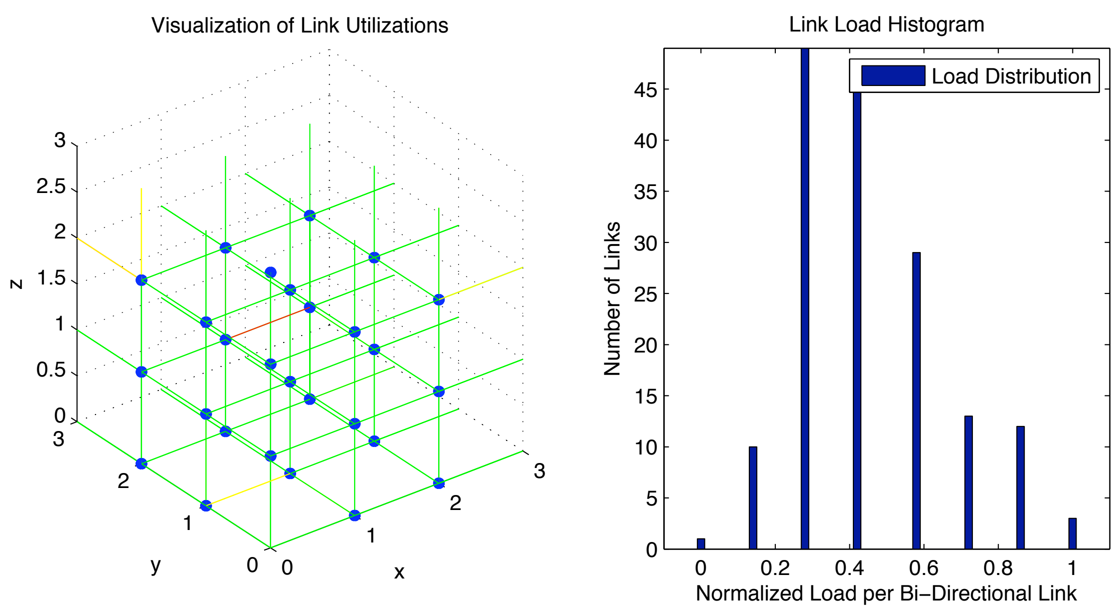


Figure E.36: *Results of Enhanced CQR Routing using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

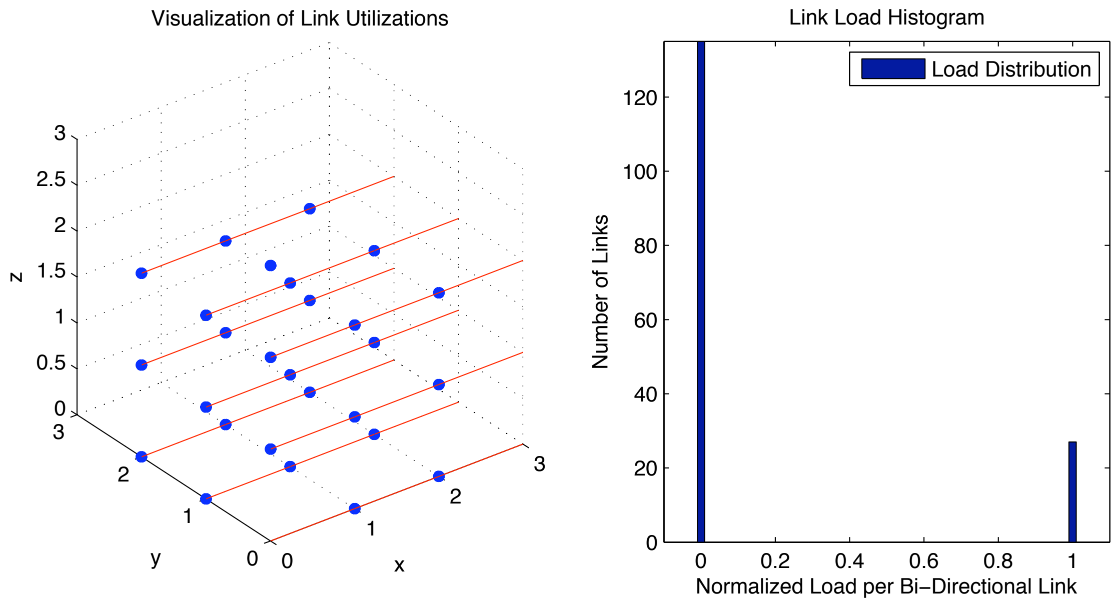


Figure E.37: *Results of Enhanced CQR Routing using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

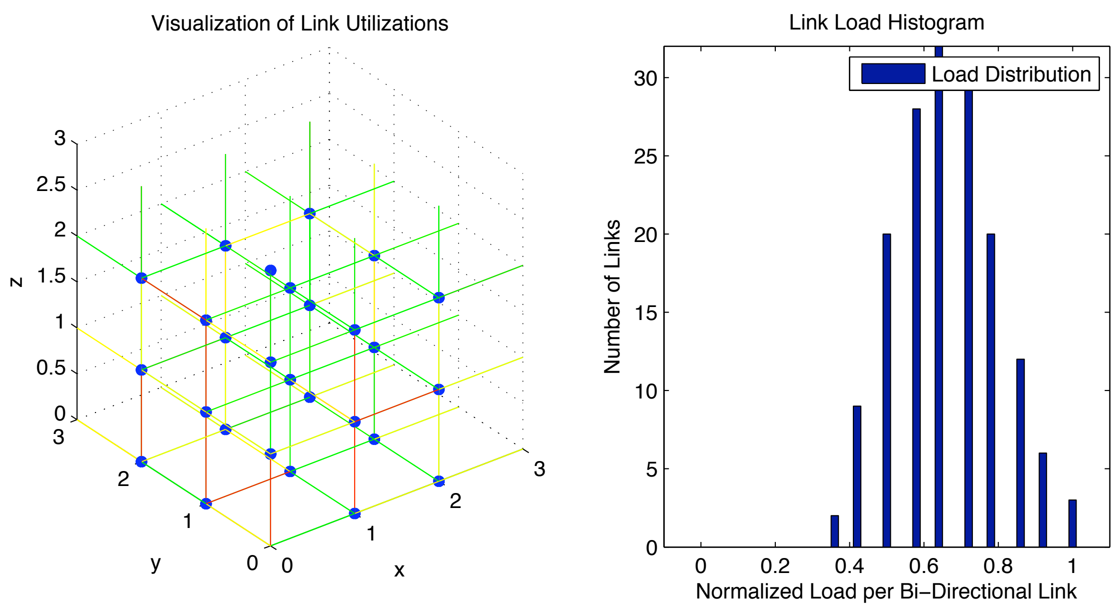


Figure E.38: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

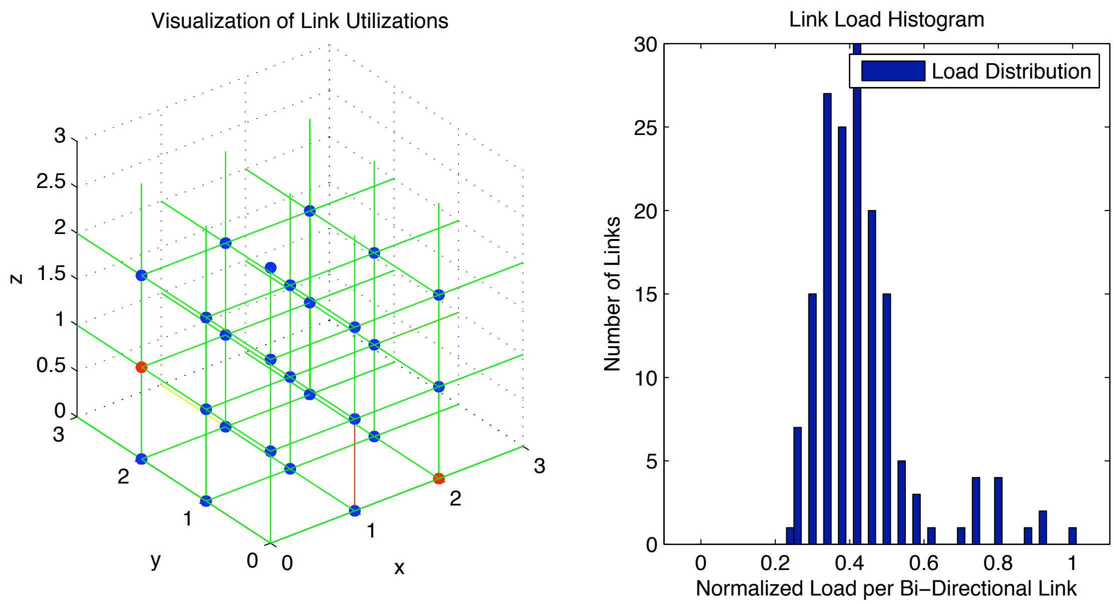


Figure E.39: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

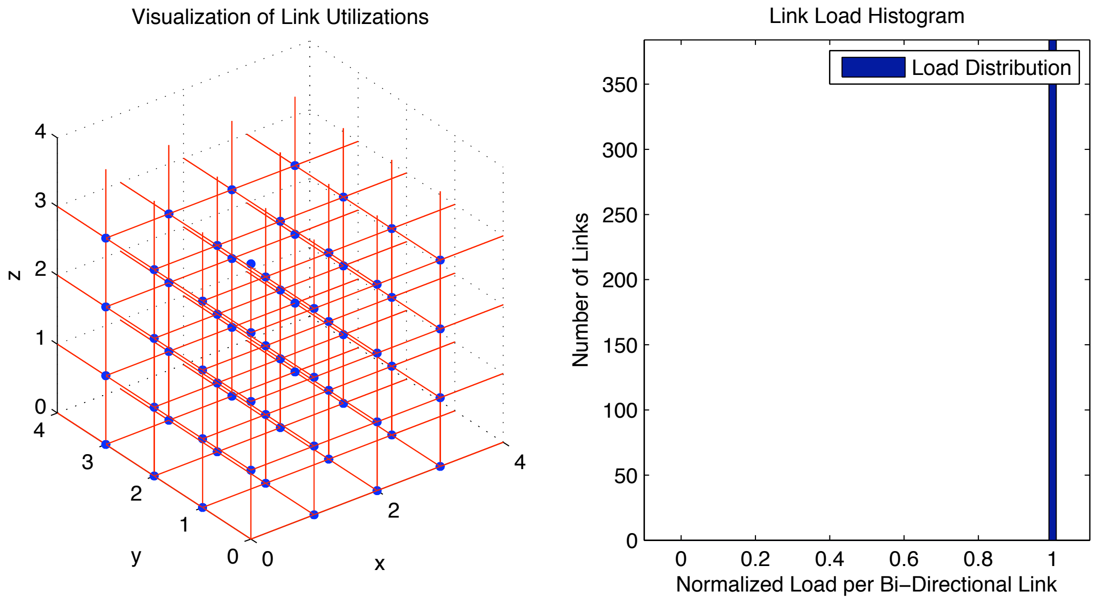


Figure E.40: *Results of Dimension Ordered Routing (DOR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

E.2 Results from the 4-ary 3-cube Topology

E.2.1 Static Algorithms

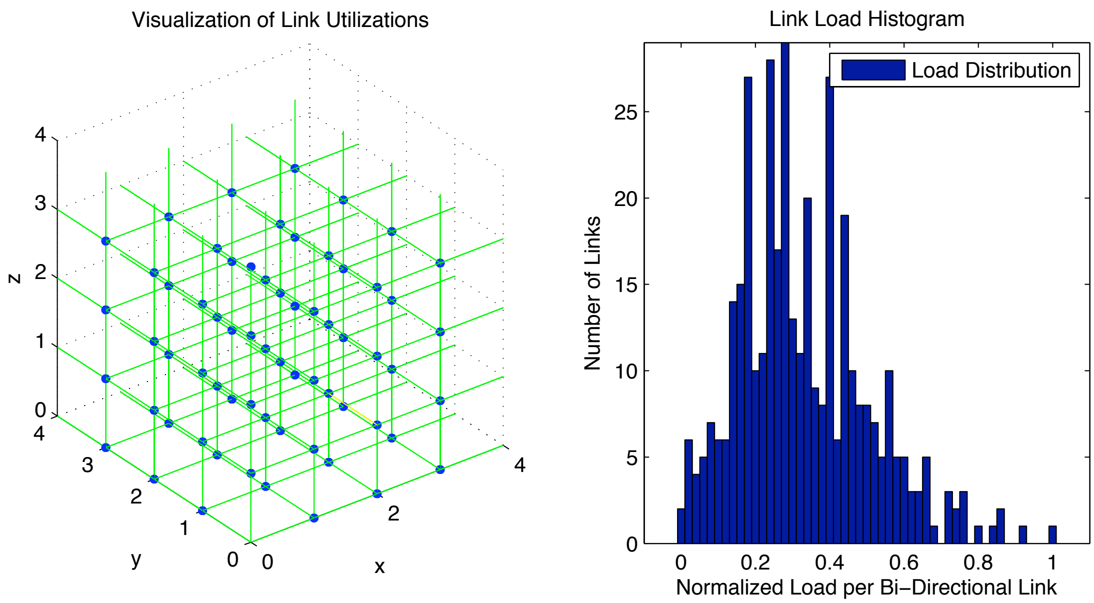


Figure E.41: *Results of Dimension Ordered Routing (DOR) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

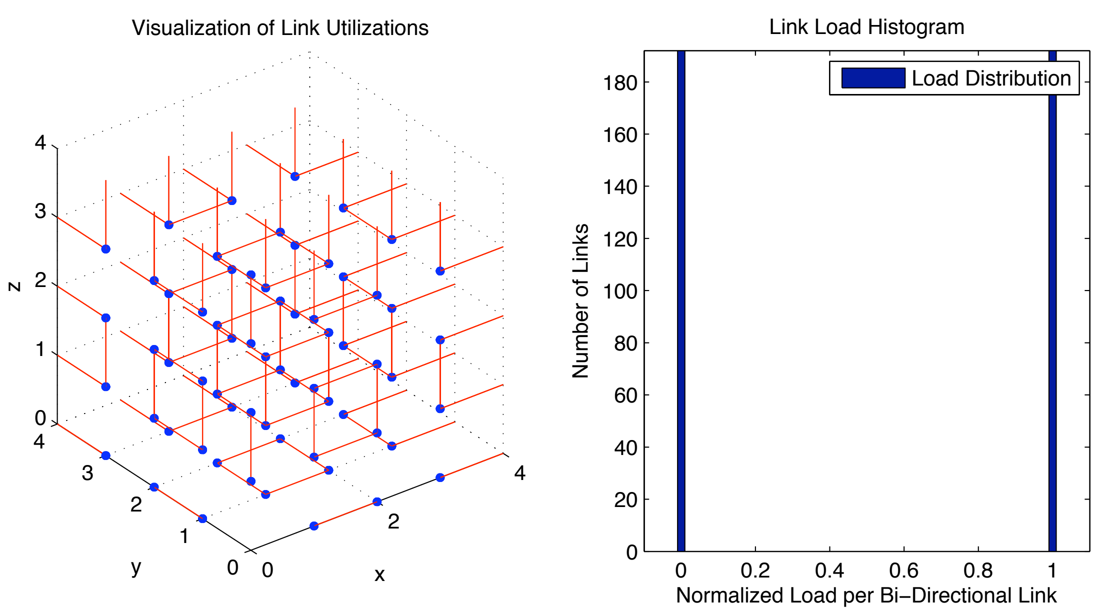


Figure E.42: *Results of Dimension Ordered Routing (DOR) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

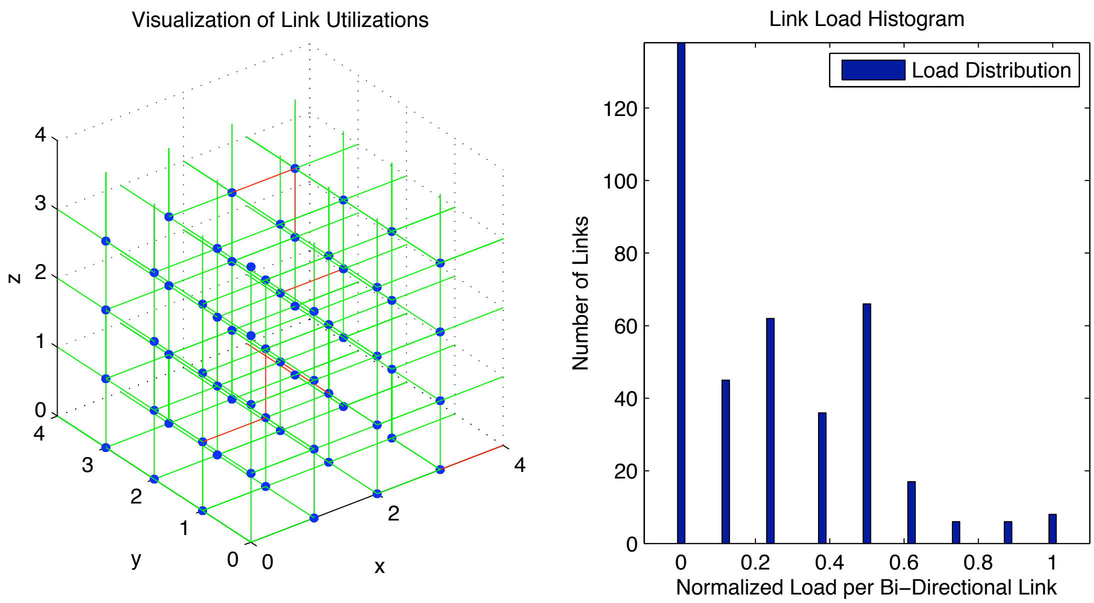


Figure E.43: *Results of Dimension Ordered Routing (DOR) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

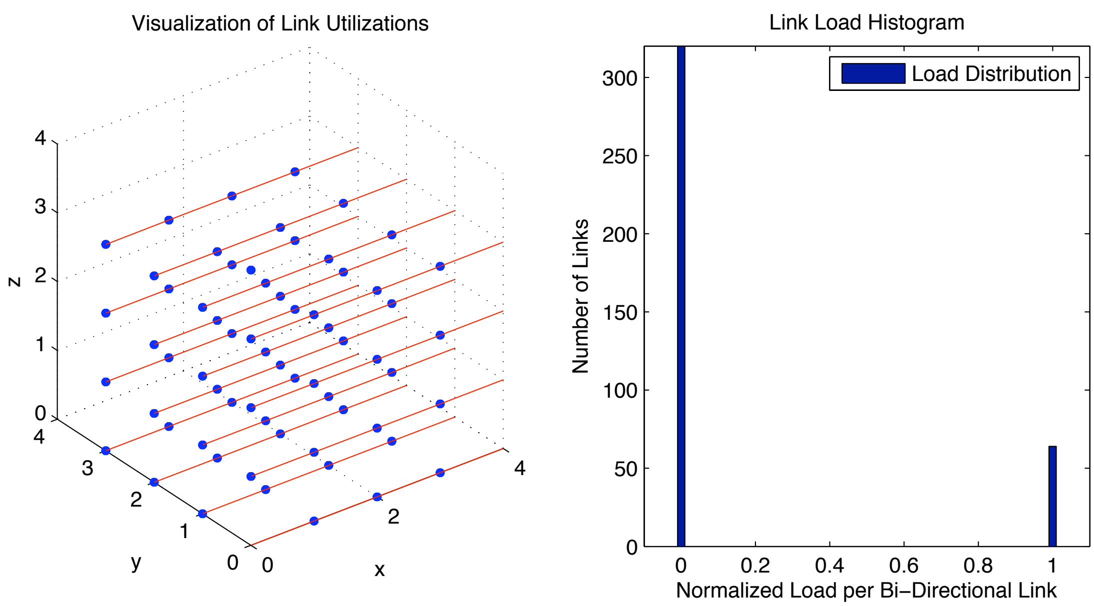


Figure E.44: *Results of Dimension Ordered Routing (DOR) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

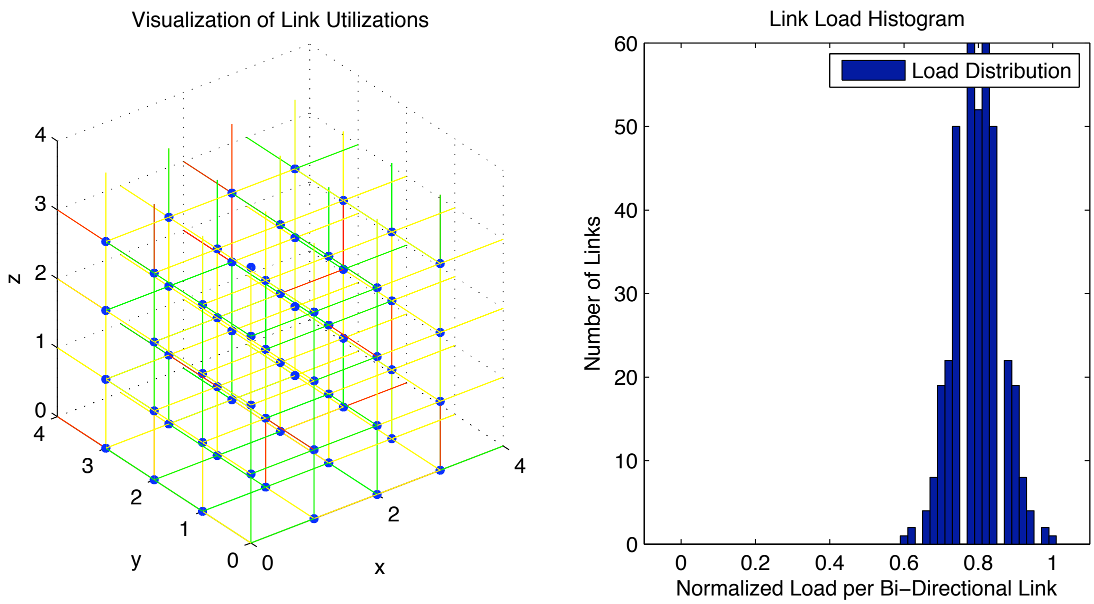


Figure E.45: *Results of Dimension Ordered Routing (DOR) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

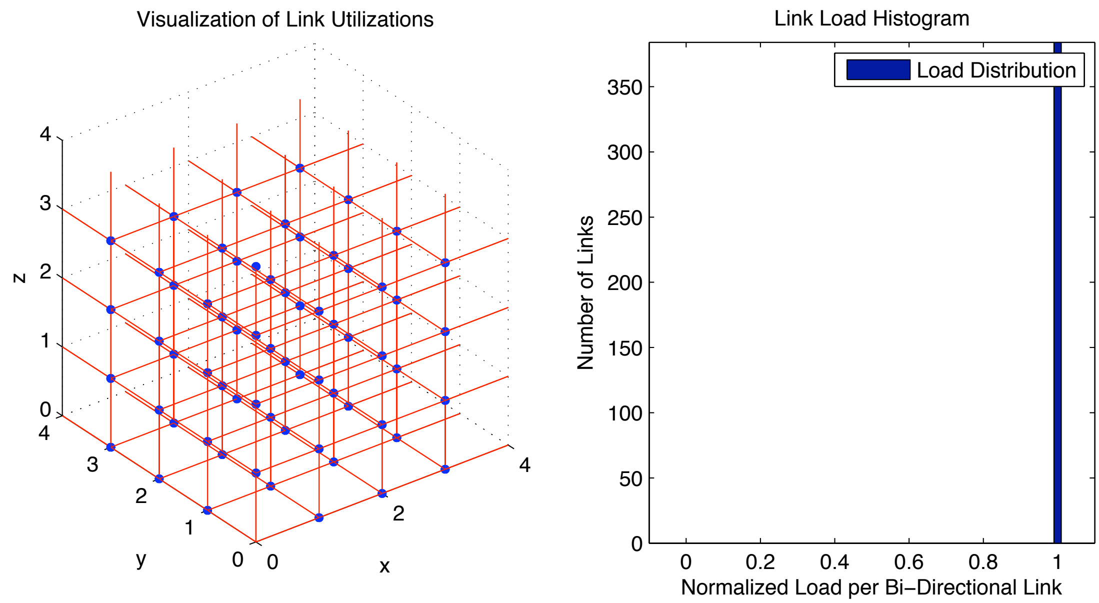


Figure E.46: *Results of Direction Ordered Routing (DIR) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

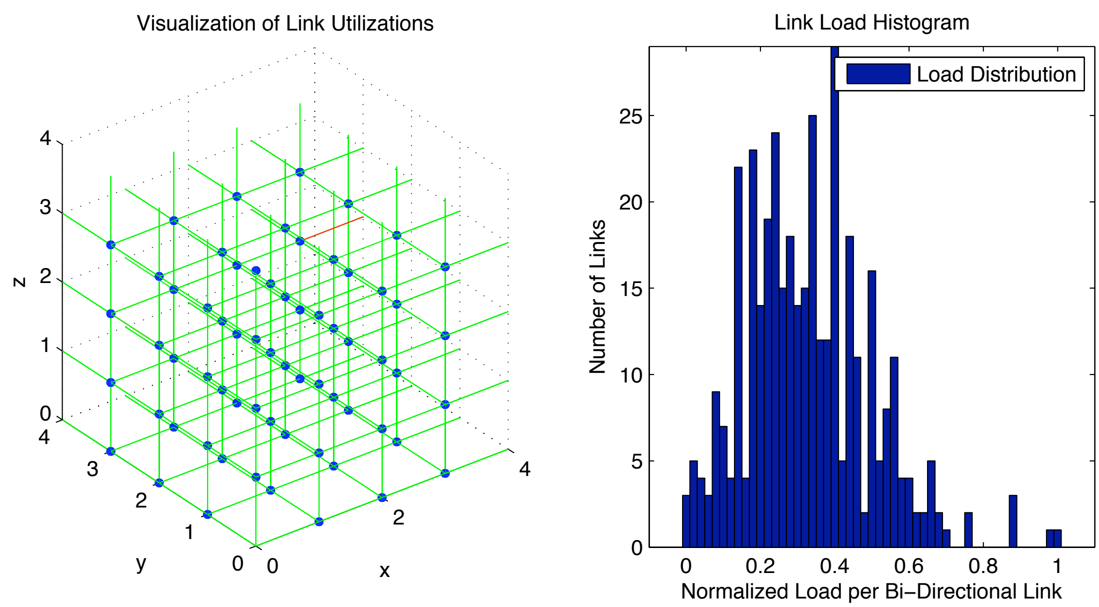


Figure E.47: *Results of Direction Ordered Routing (DIR) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

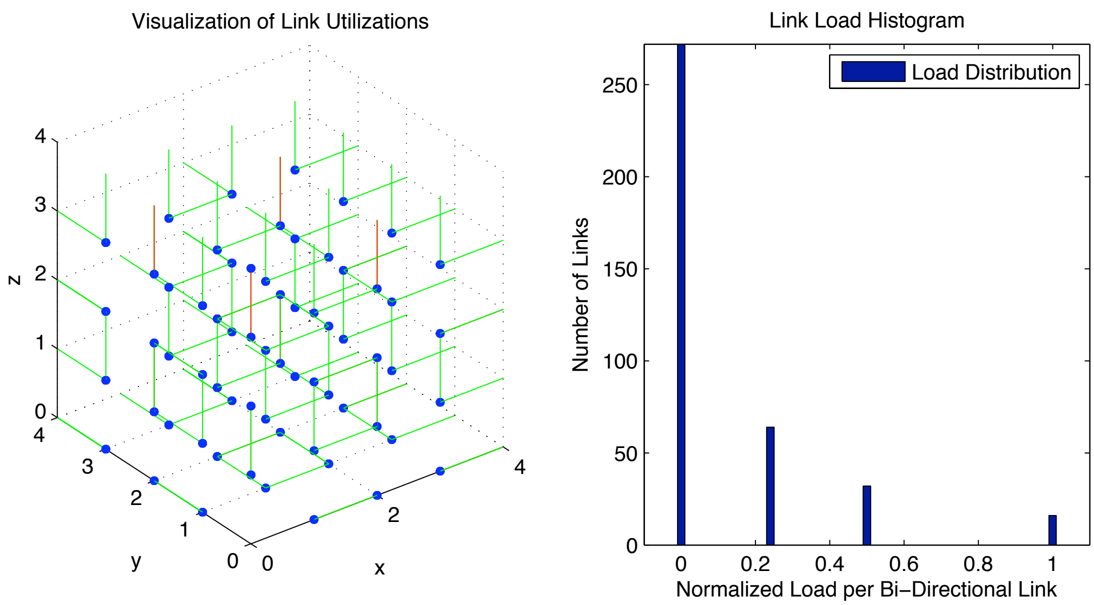


Figure E.48: *Results of Direction Ordered Routing (DIR) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

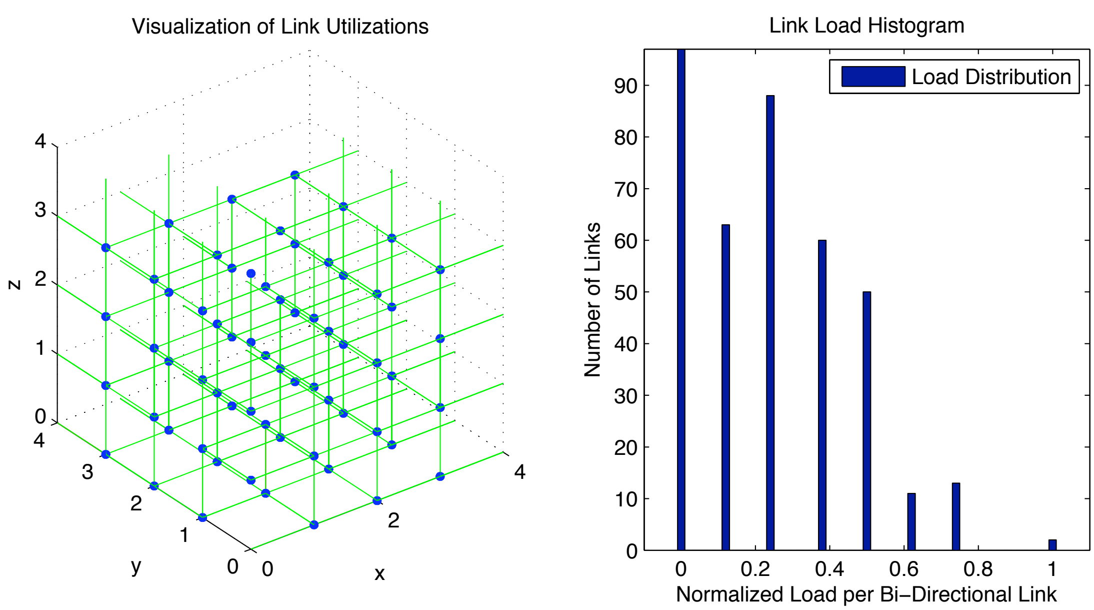


Figure E.49: *Results of Direction Ordered Routing (DIR) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

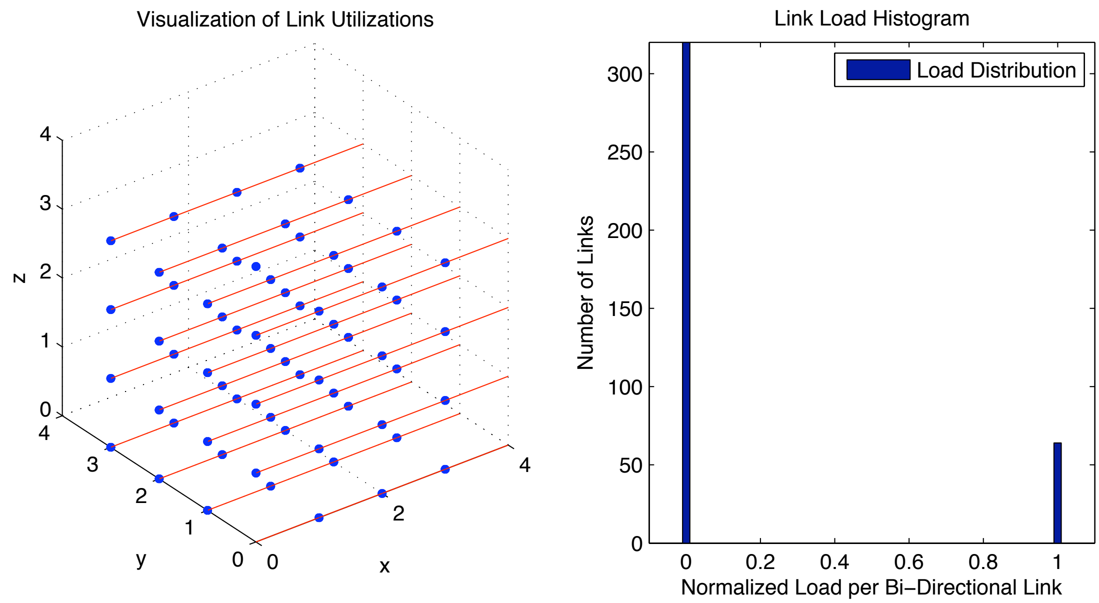


Figure E.50: *Results of Direction Ordered Routing (DIR) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

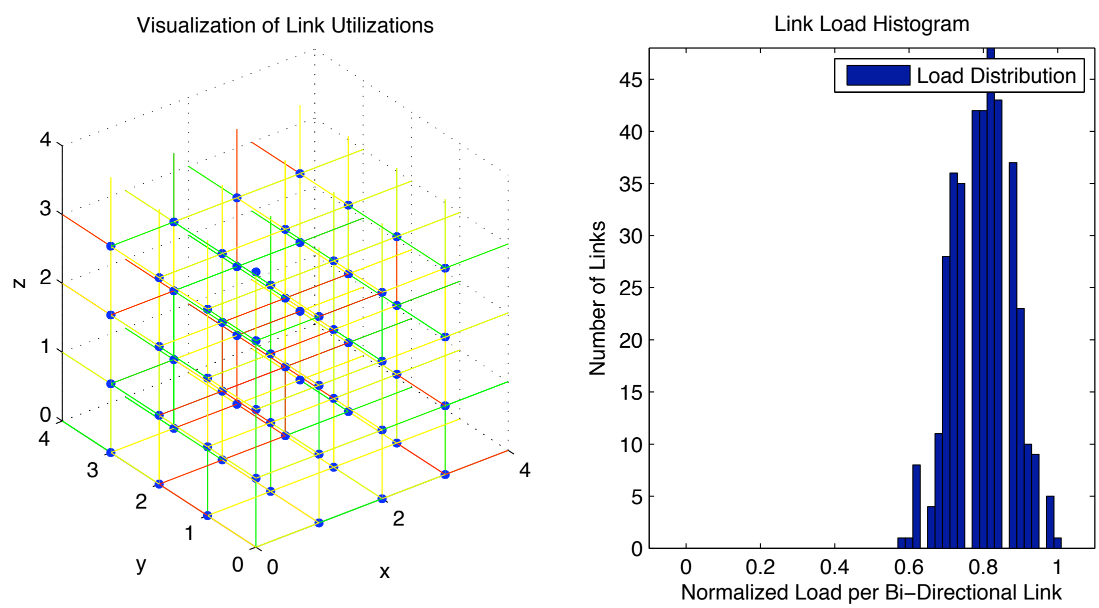


Figure E.51: *Results of Direction Ordered Routing (DIR) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

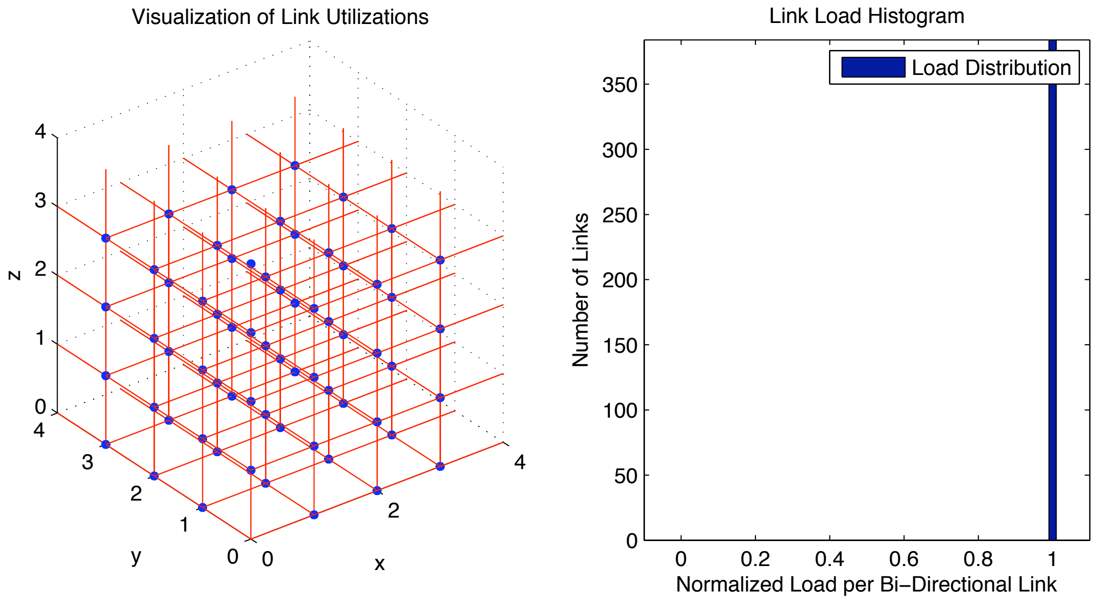


Figure E.52: *Results of Minimal Oblivious Routing (MO) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

E.2.2 Oblivious Algorithms

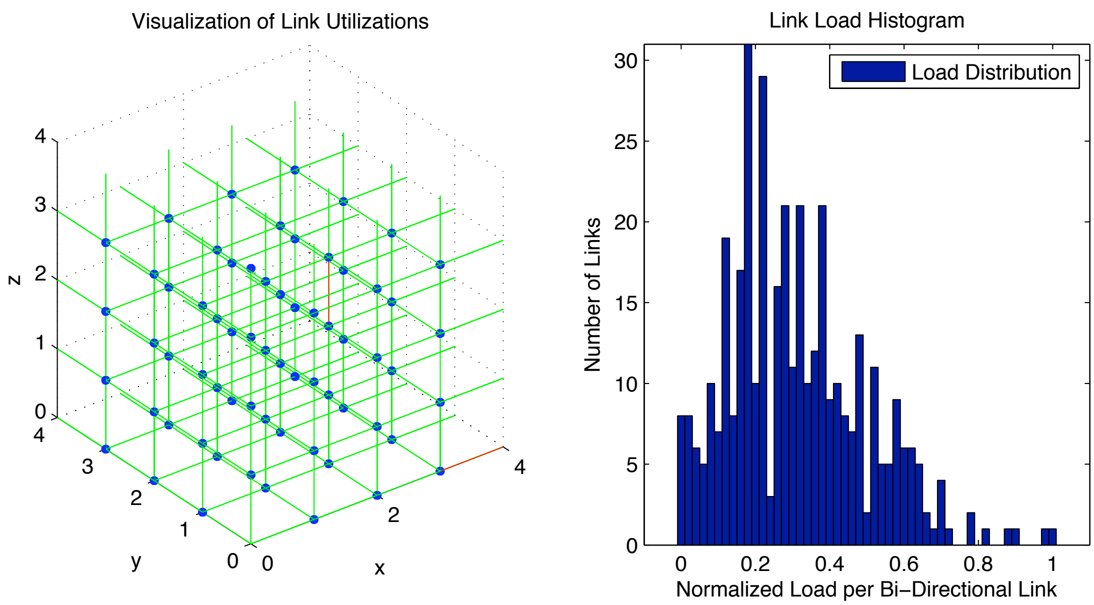


Figure E.53: *Results of Minimal Oblivious Routing (MO) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

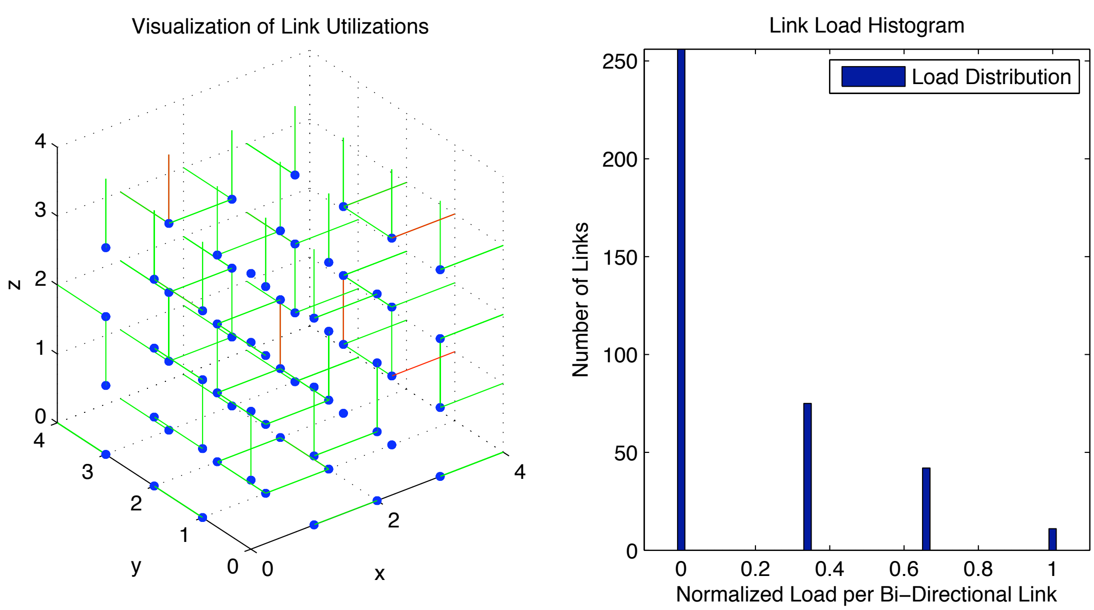


Figure E.54: *Results of Minimal Oblivious Routing (MO) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

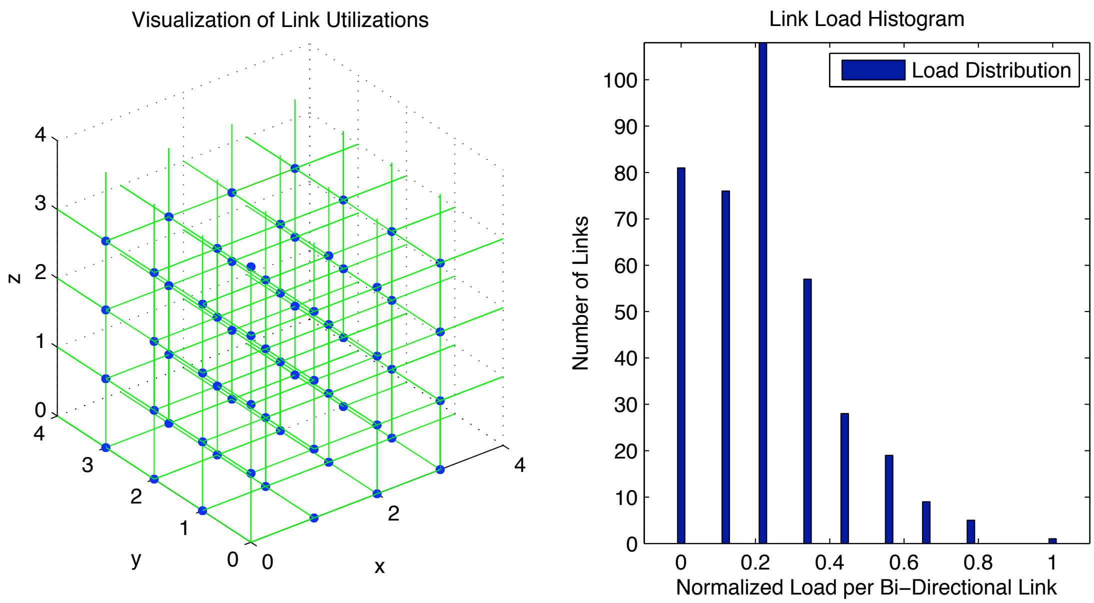


Figure E.55: *Results of Minimal Oblivious Routing (MO) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

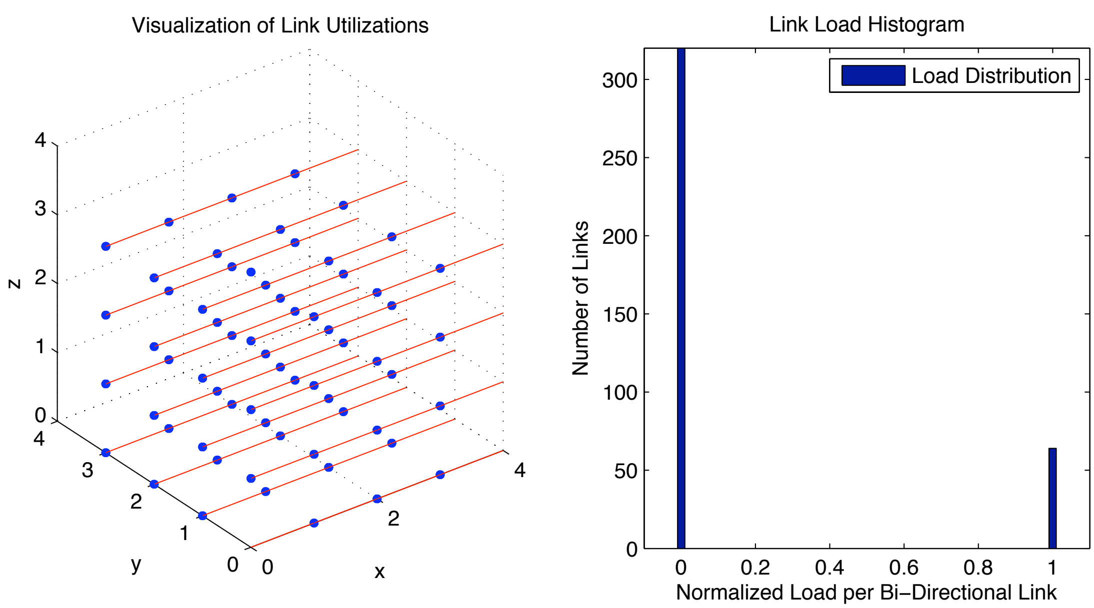


Figure E.56: *Results of Minimal Oblivious Routing (MO) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

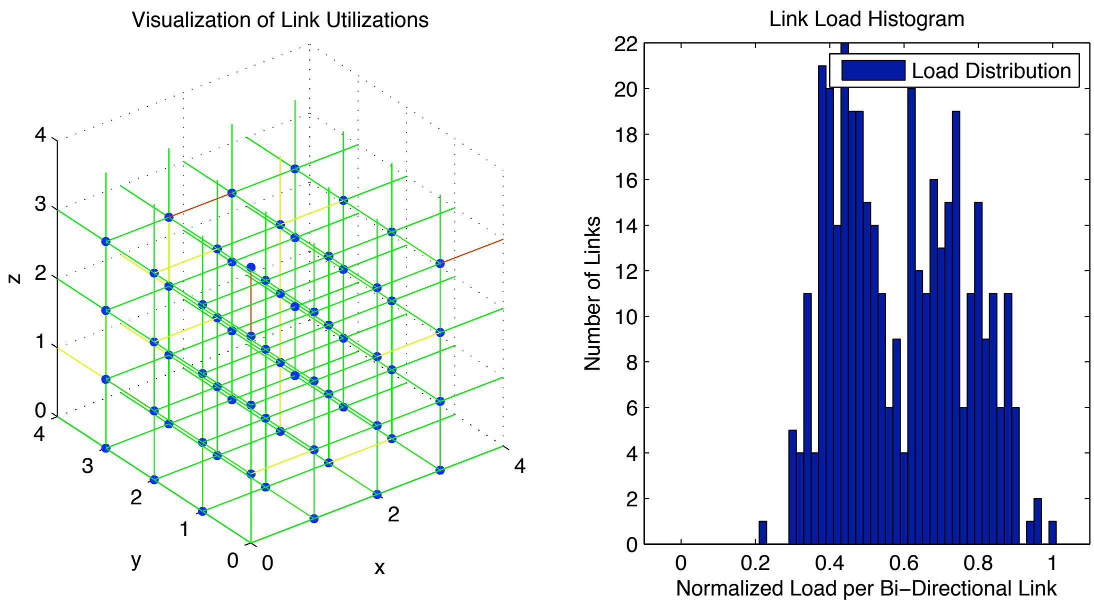


Figure E.57: *Results of Minimal Oblivious Routing (MO) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

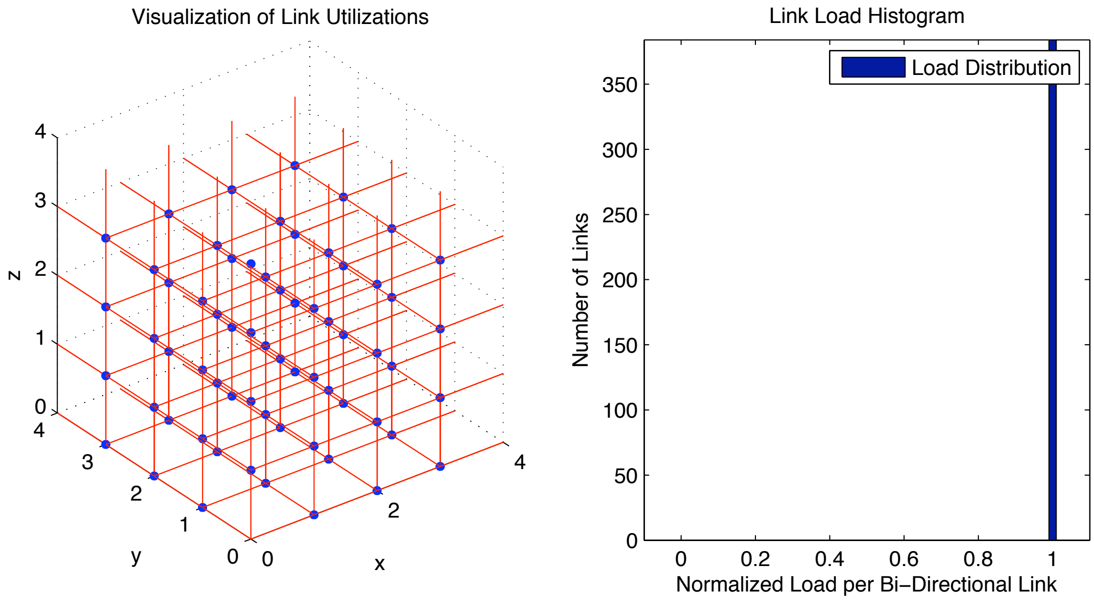


Figure E.58: *Results of Minimal Adaptive Routing (MA) using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

E.2.3 Adaptive Algorithms

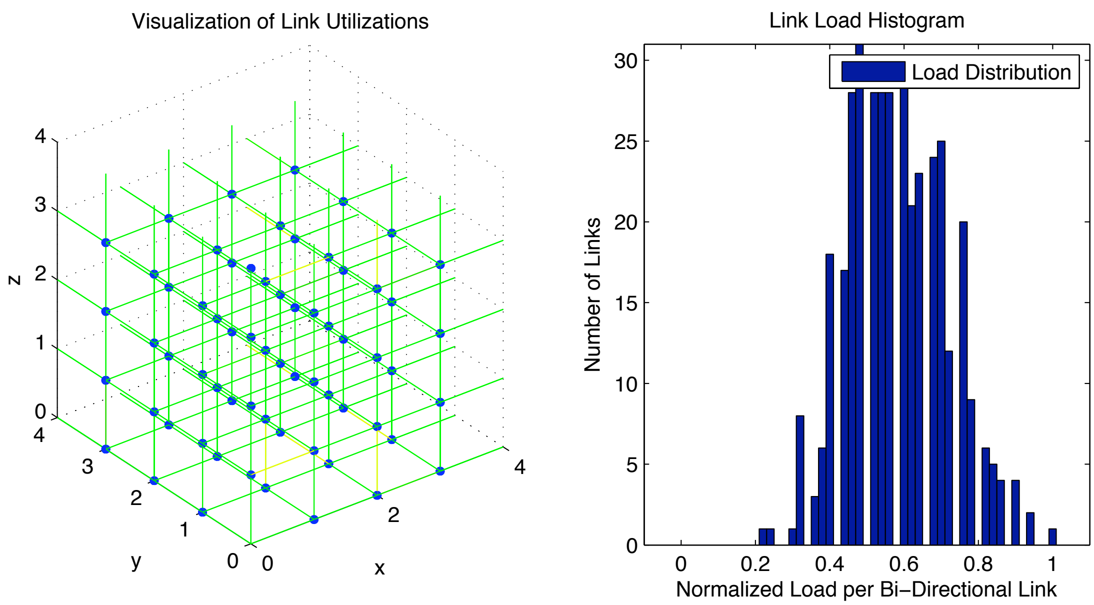


Figure E.59: *Results of Minimal Adaptive Routing (MA) using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

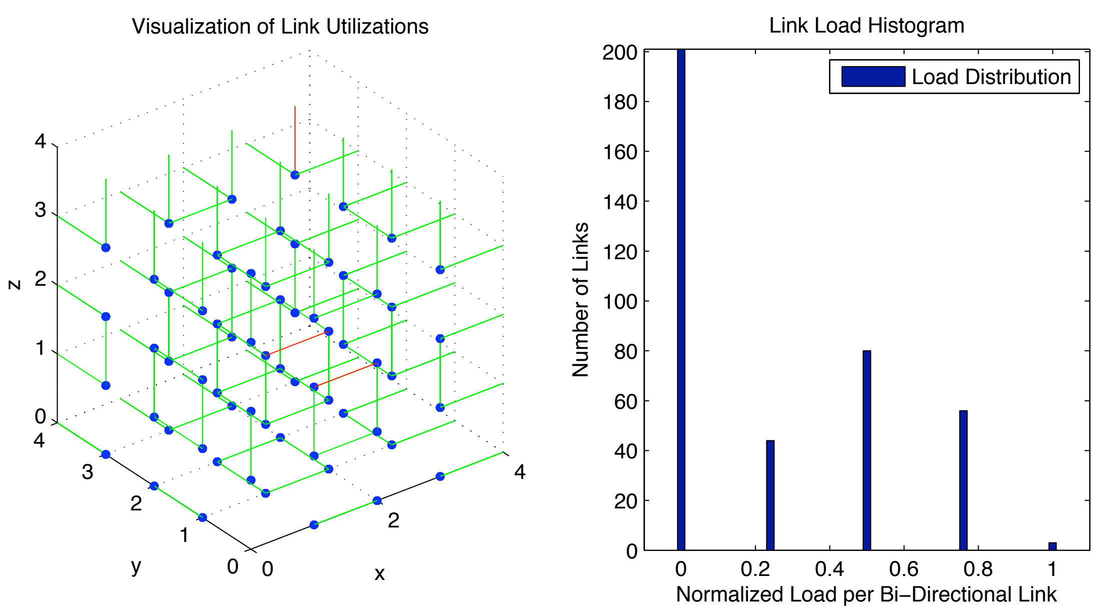


Figure E.60: *Results of Minimal Adaptive Routing (MA) using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

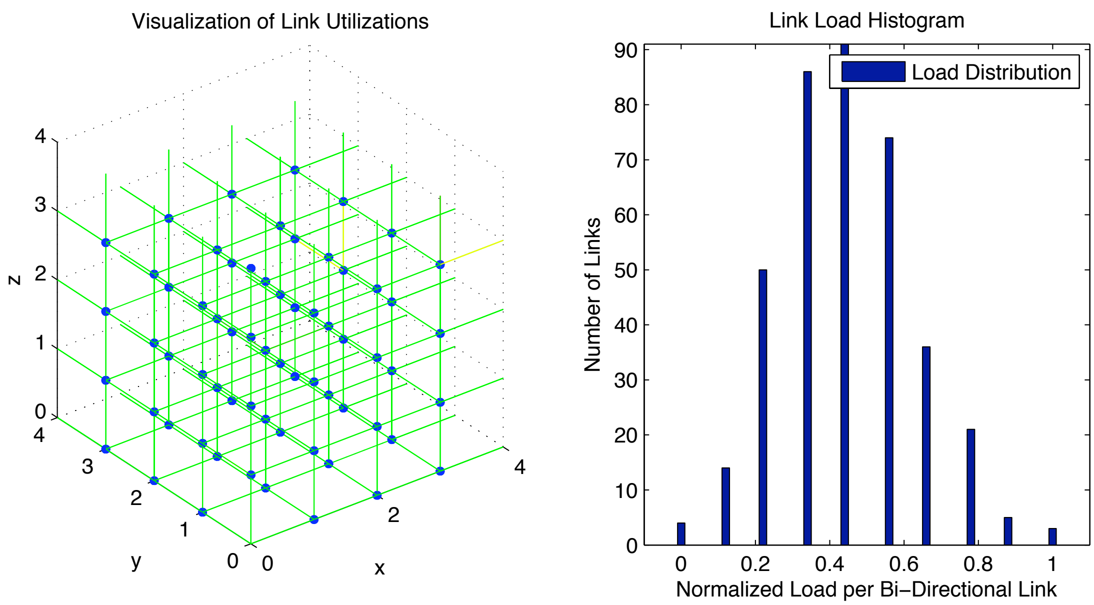


Figure E.61: *Results of Minimal Adaptive Routing (MA) using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

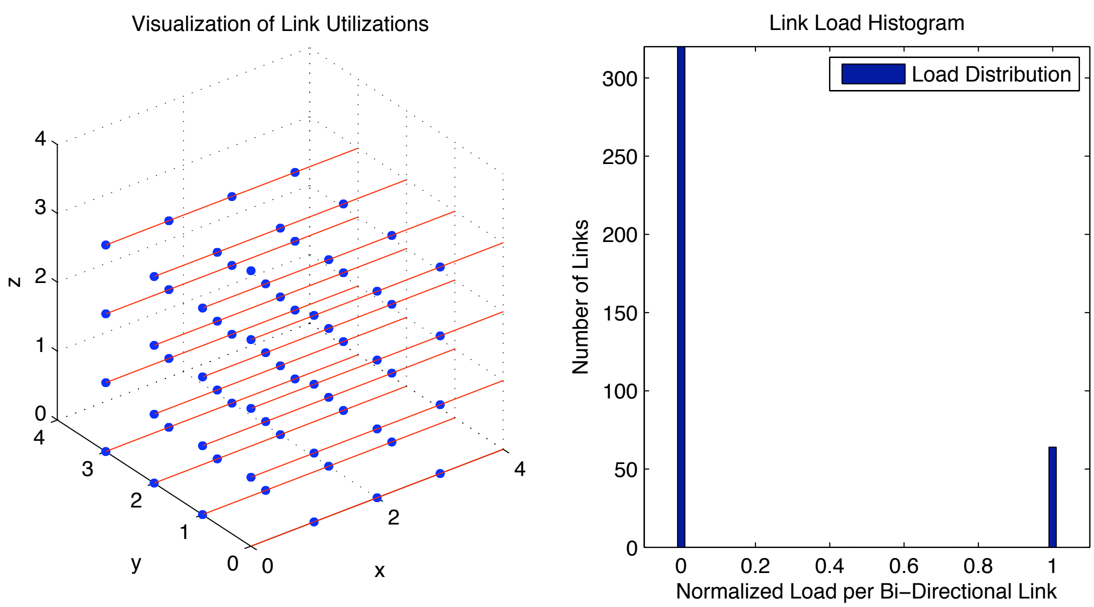


Figure E.62: *Results of Minimal Adaptive Routing (MA) using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

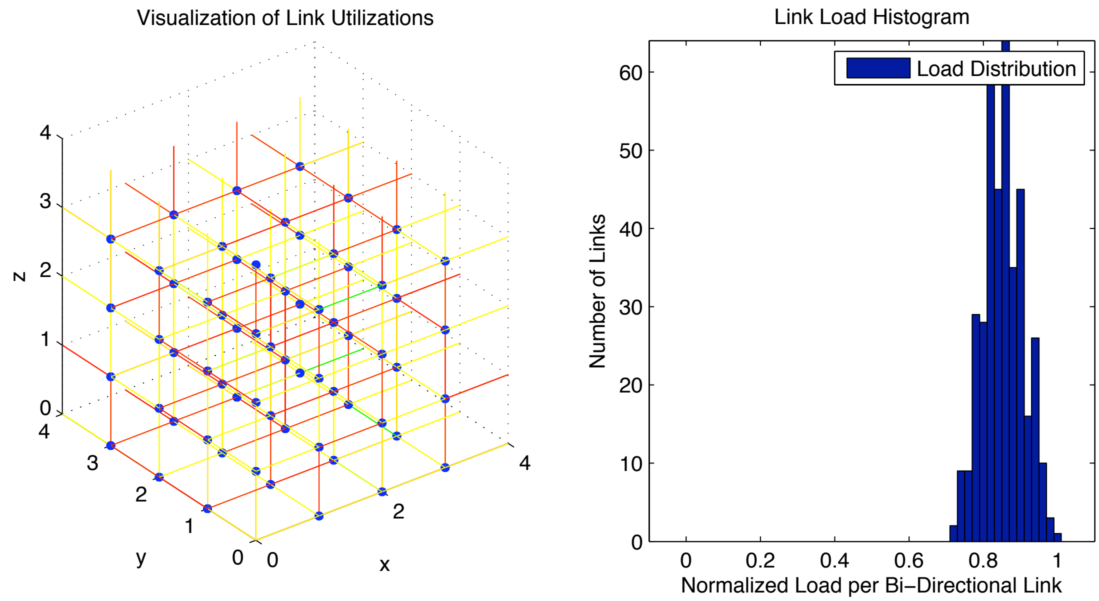


Figure E.63: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

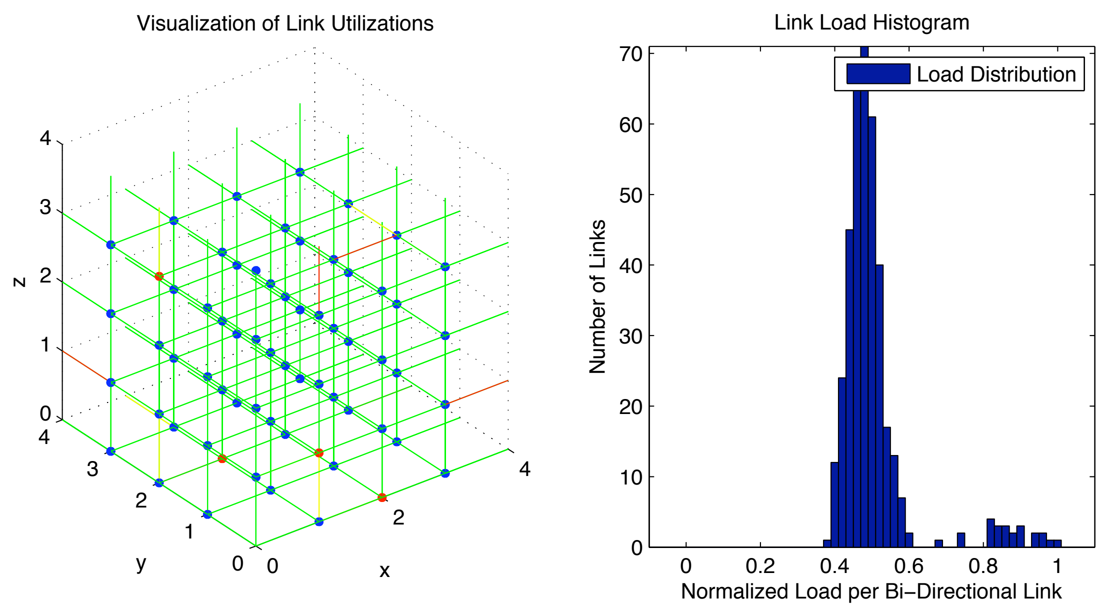


Figure E.64: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

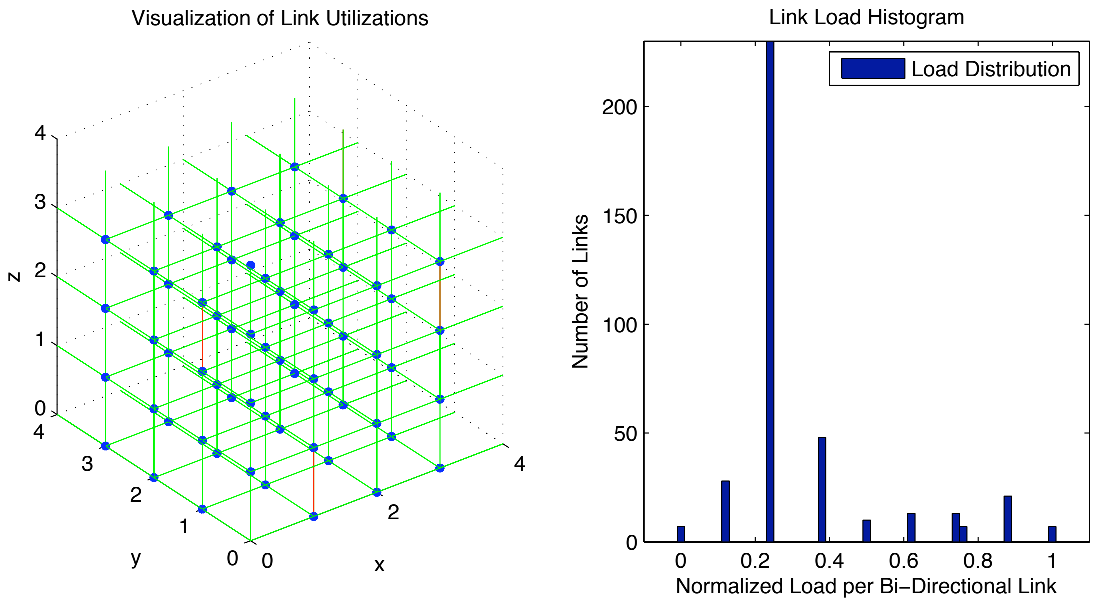


Figure E.65: *Results of CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

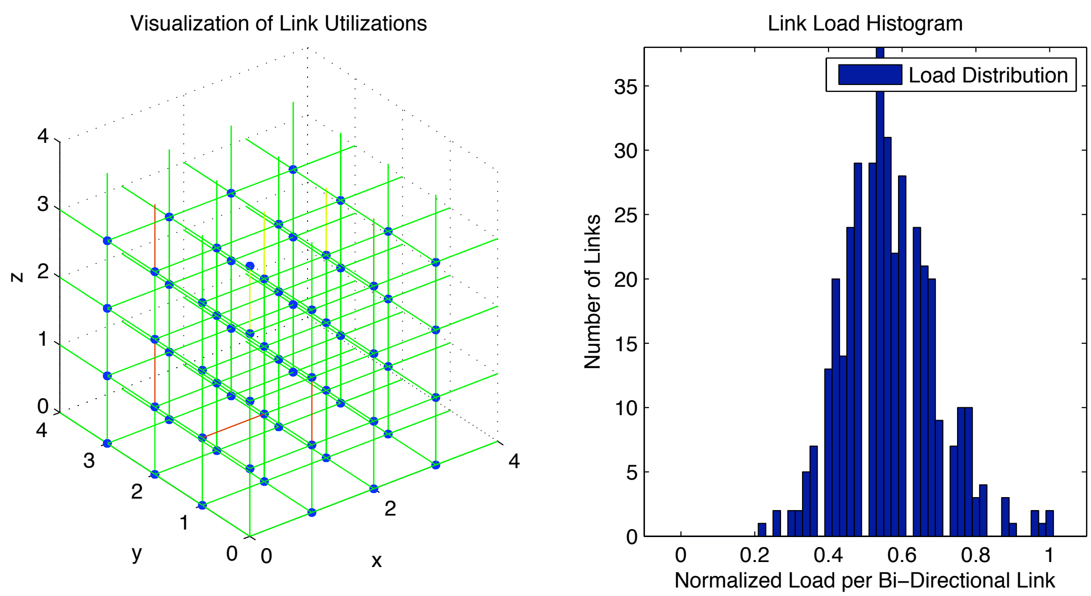


Figure E.66: *Results of CQR Routing using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

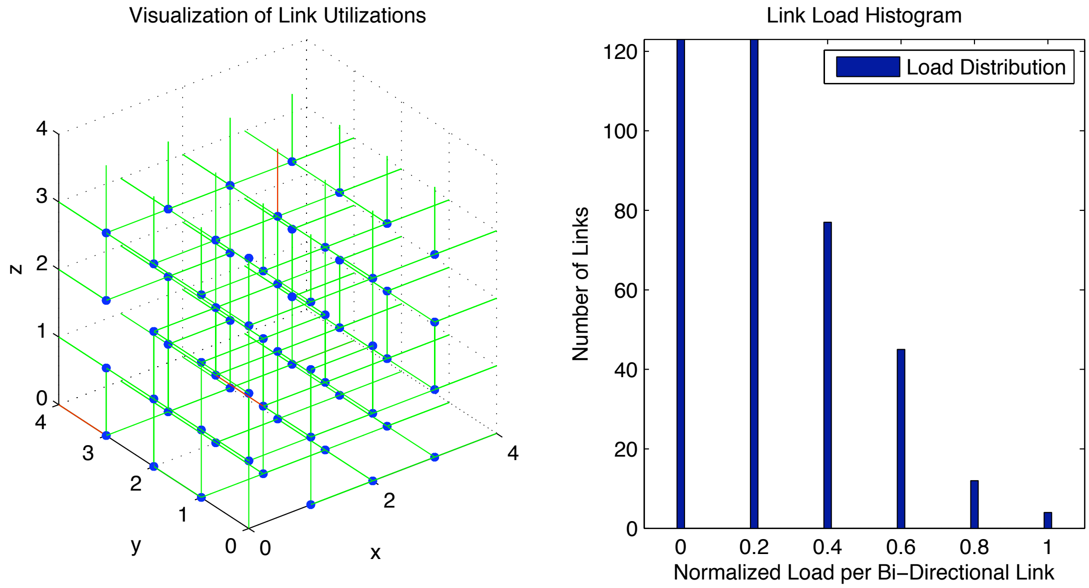


Figure E.67: *Results of CQR Routing using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

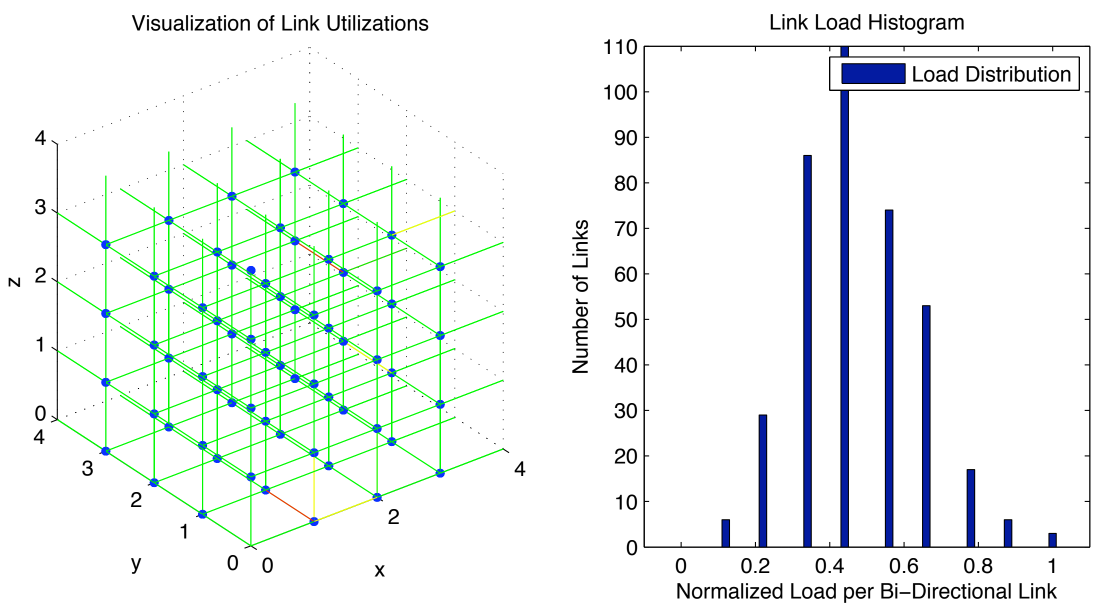


Figure E.68: *Results of CQR Routing using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

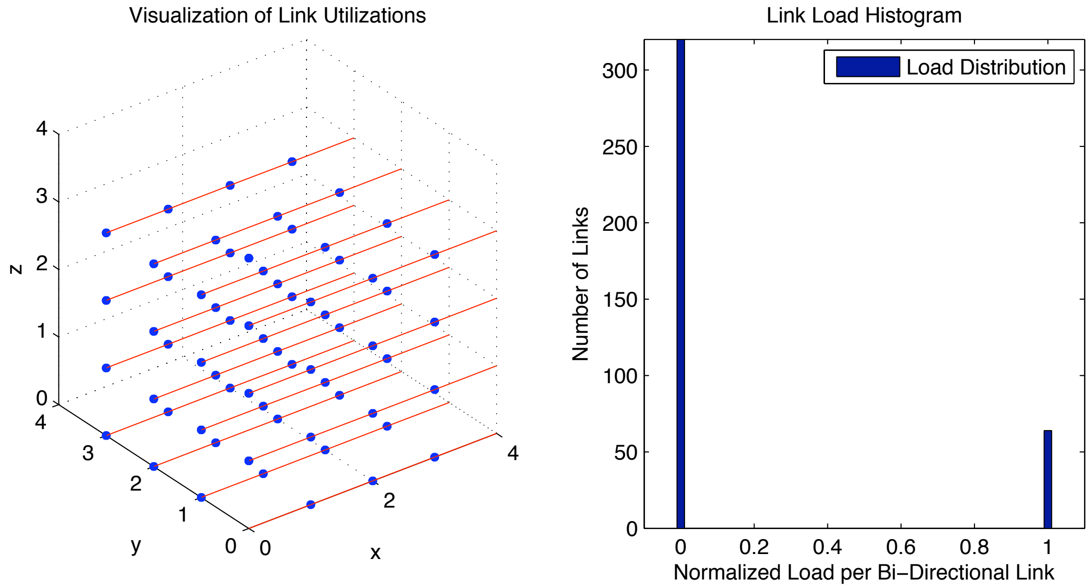


Figure E.69: *Results of CQR Routing using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

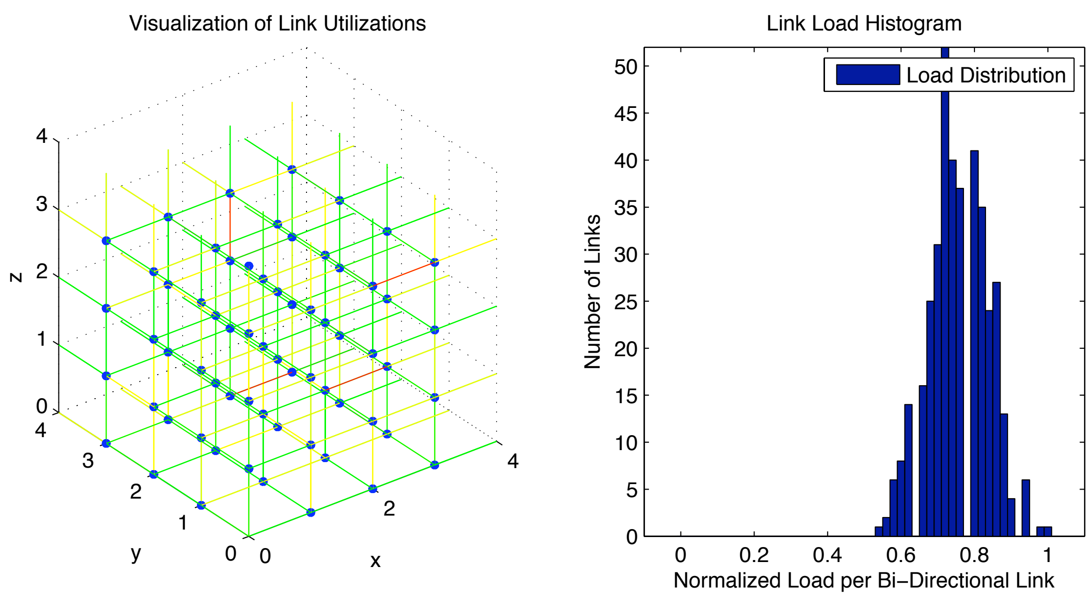


Figure E.70: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

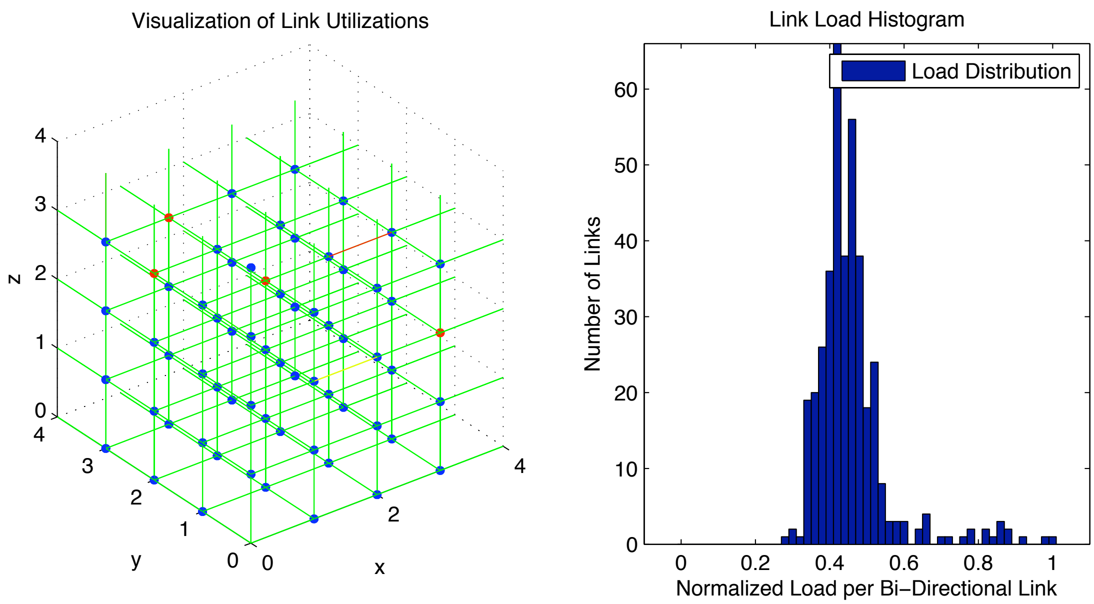


Figure E.71: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

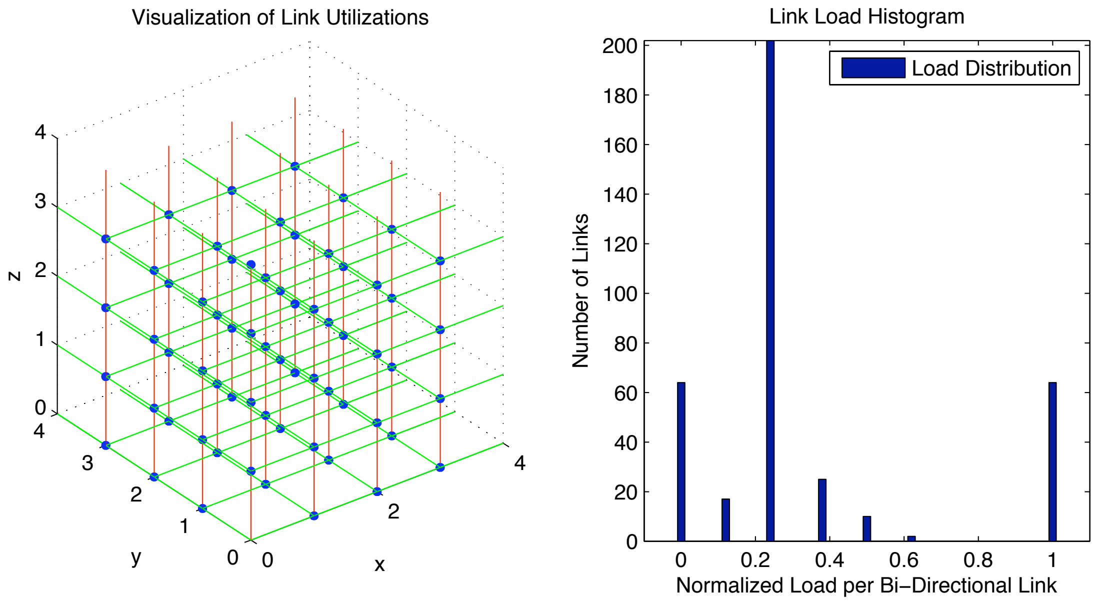


Figure E.72: *Results of Enhanced CQR Routing using the nearest neighbor traffic demand (NN). Each node had two individual traffic demands to each of its six neighbors.*

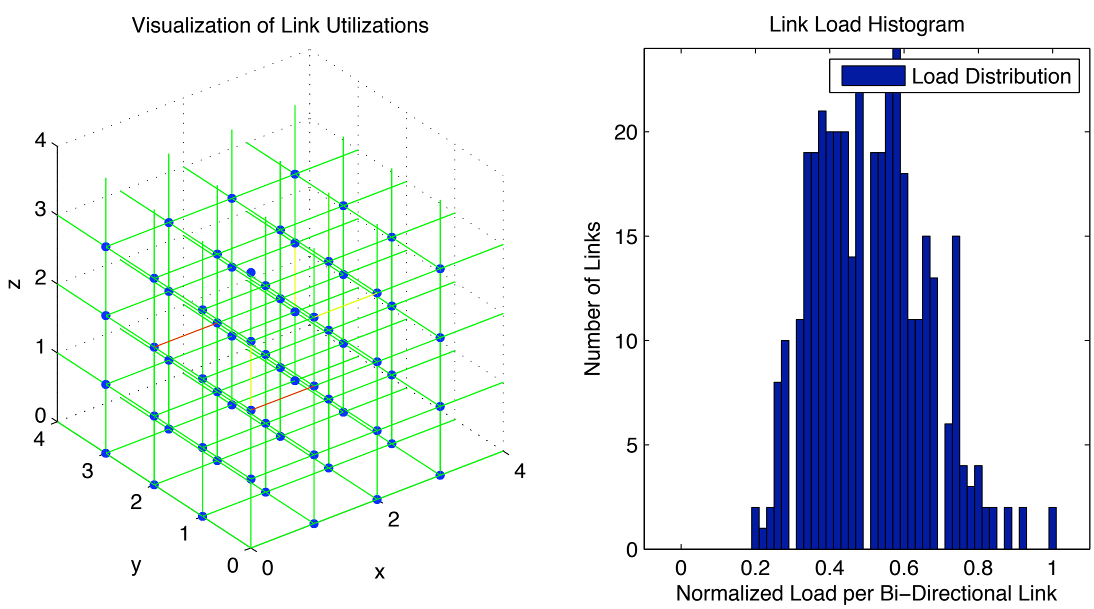


Figure E.73: *Results of Enhanced CQR Routing using the uniform random traffic demand (UR). For each node, a value of 0-9 was assigned as having 0-9 individual traffic demands.*

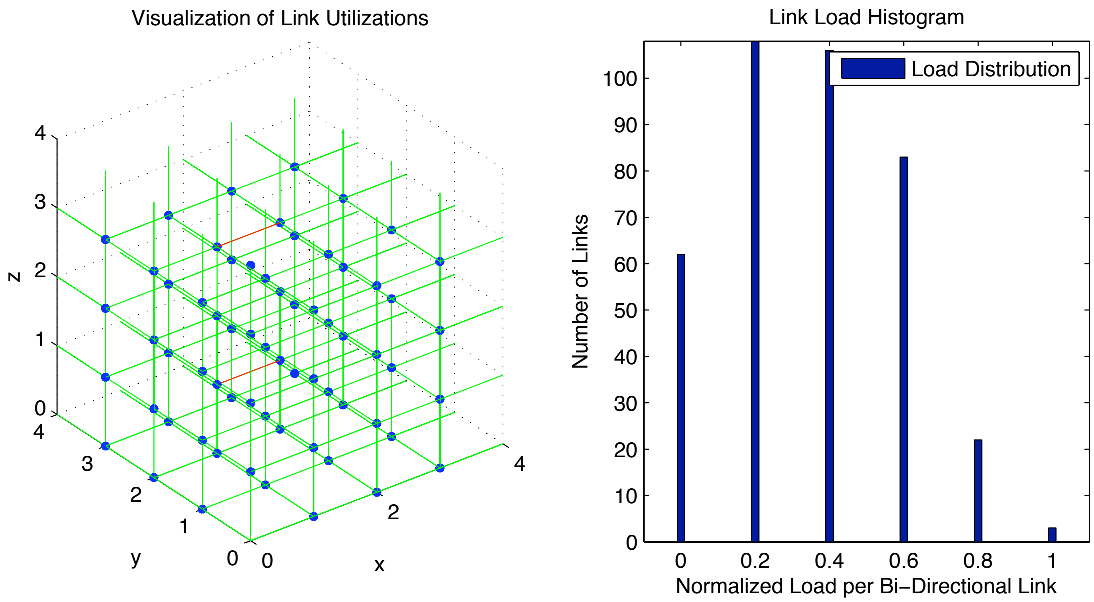


Figure E.74: *Results of CQR Routing using the bit complement traffic demand (BC). Two individual traffic demands were assigned for possible values described in Table 2.1.*

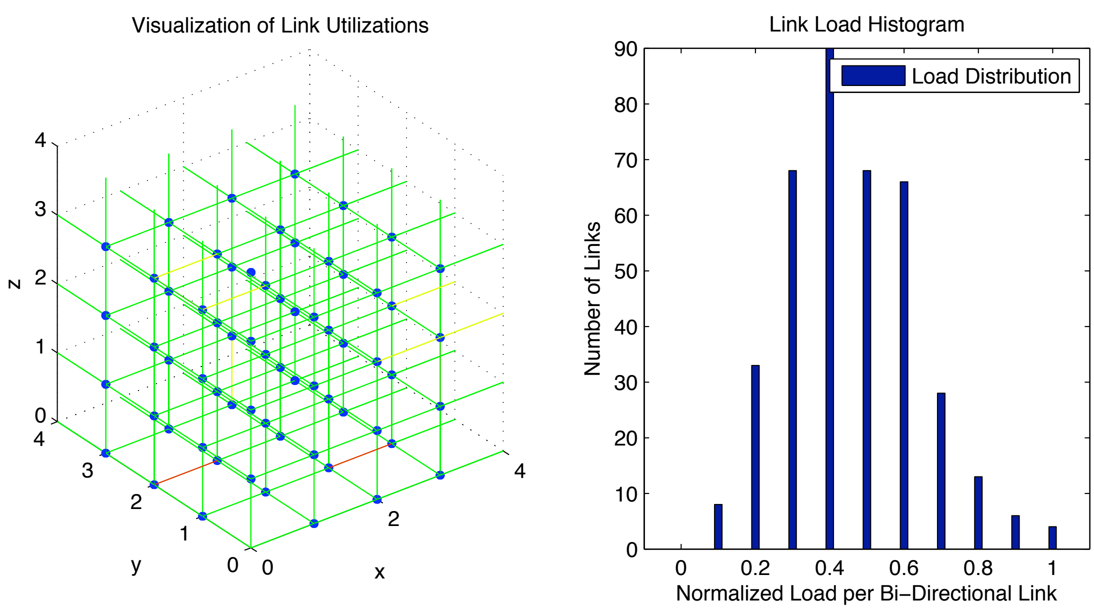


Figure E.75: *Results of Enhanced CQR Routing using the transpose traffic demand (TP). Two individual traffic demands were assigned for each permutation of all x,y,z values.*

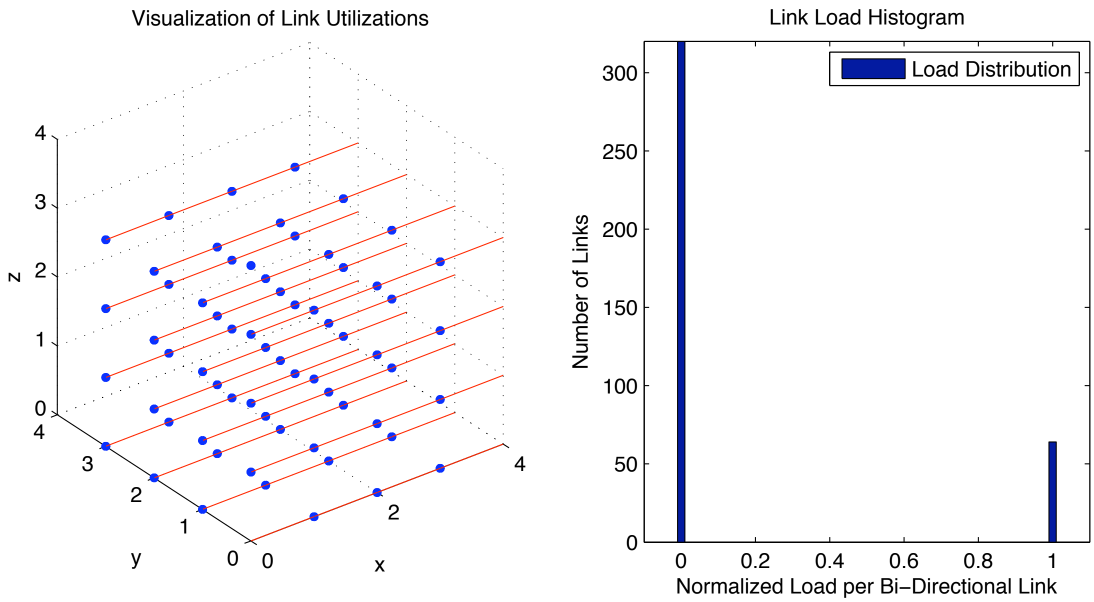


Figure E.76: *Results of Enhanced CQR Routing using the tornado traffic demand (TOR). Three individual traffic demands were assigned for possible values described in Table 2.1.*

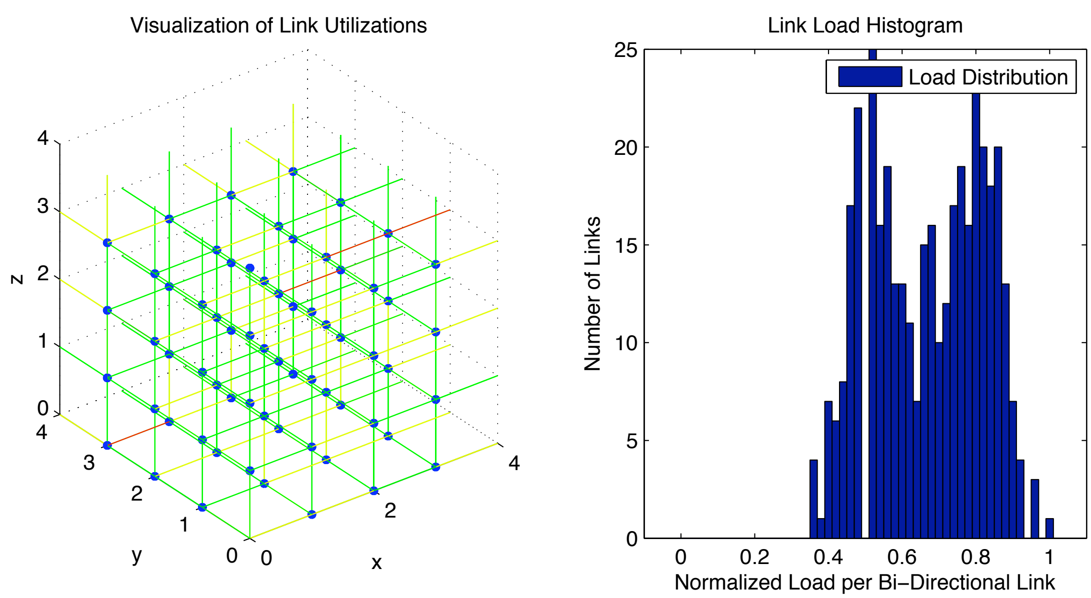


Figure E.77: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

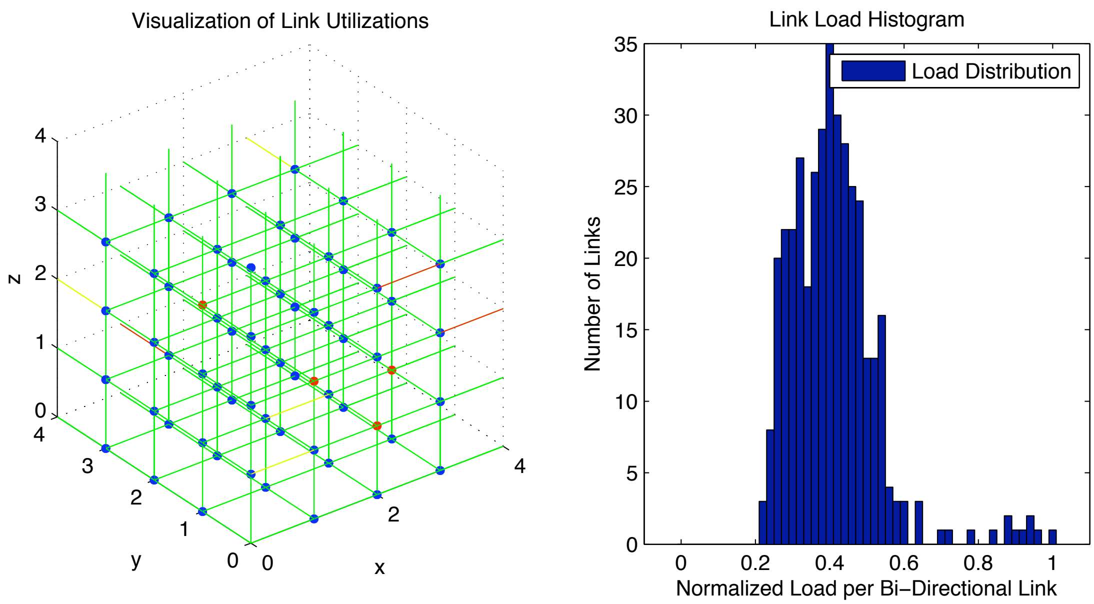


Figure E.78: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

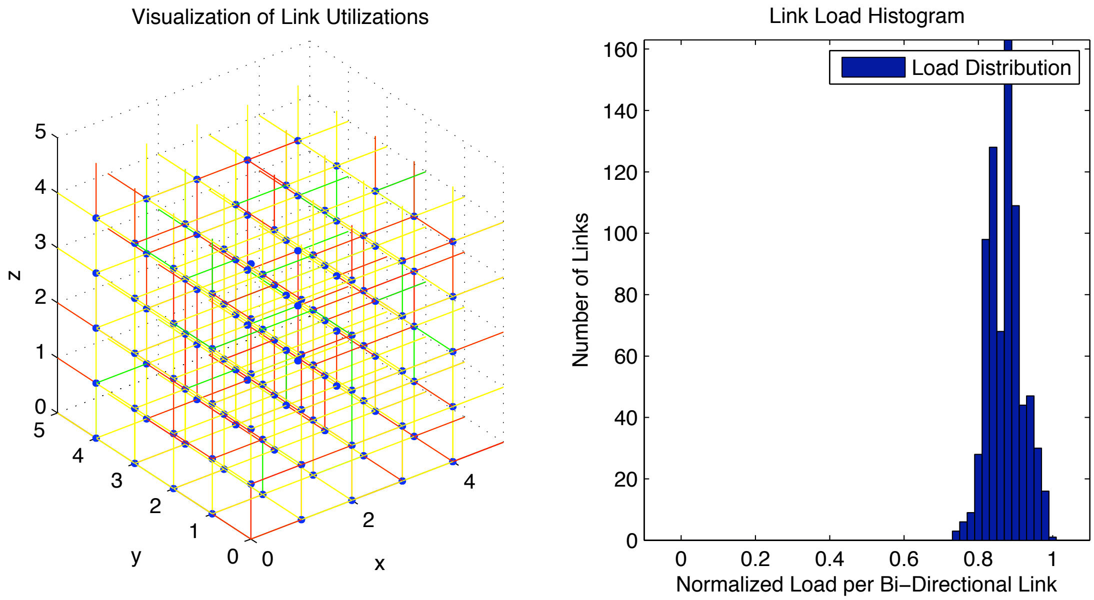


Figure E.79: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

E.3 Results from the 5-ary 3-cube Topology

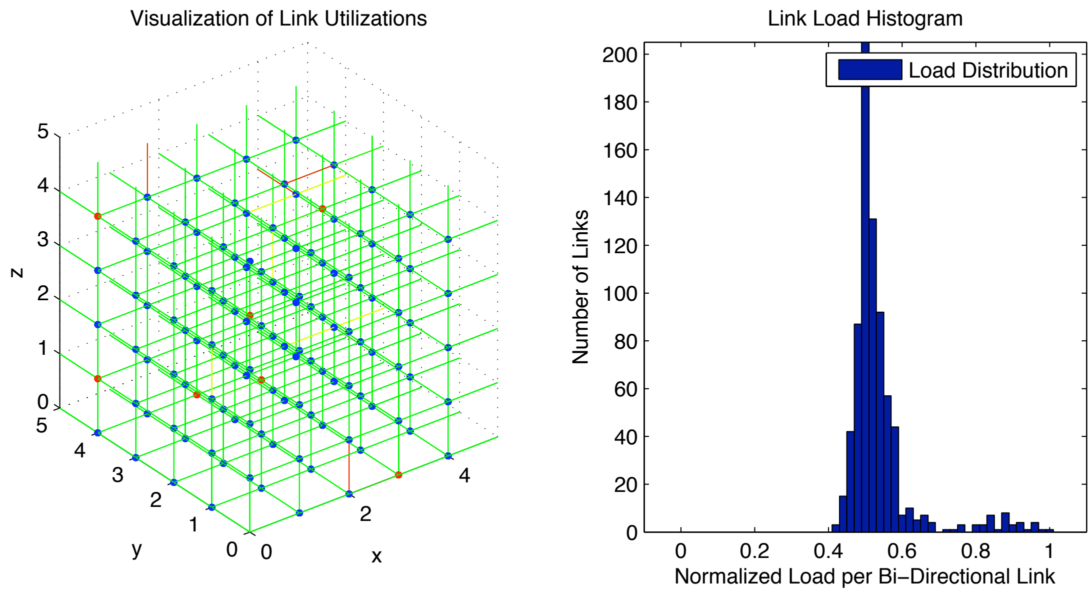


Figure E.80: *Results of Minimal Adaptive Routing (MA) using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

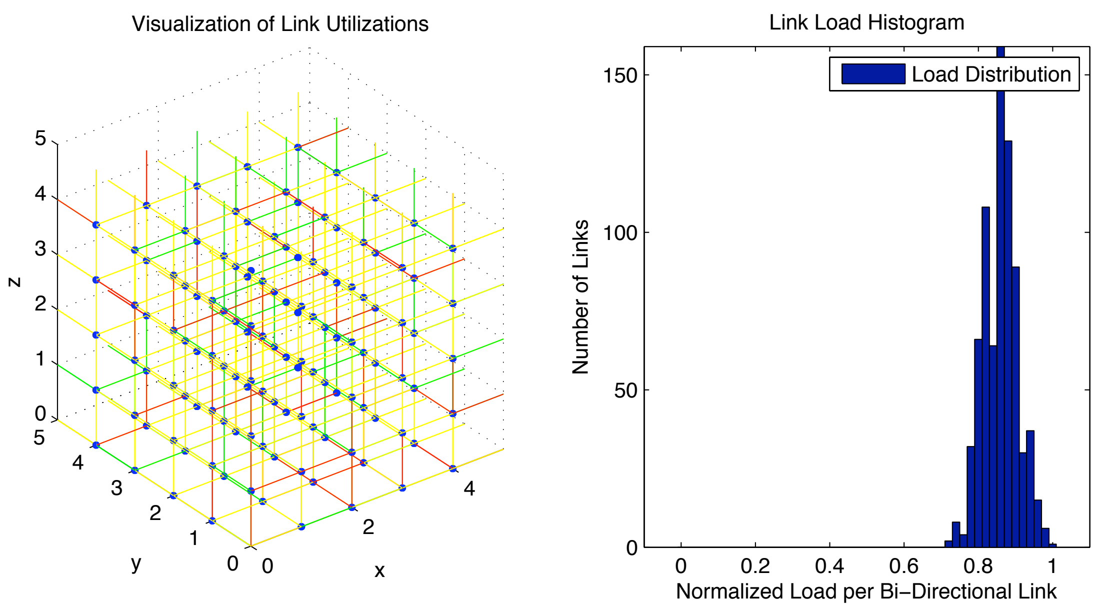


Figure E.81: *Results of CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

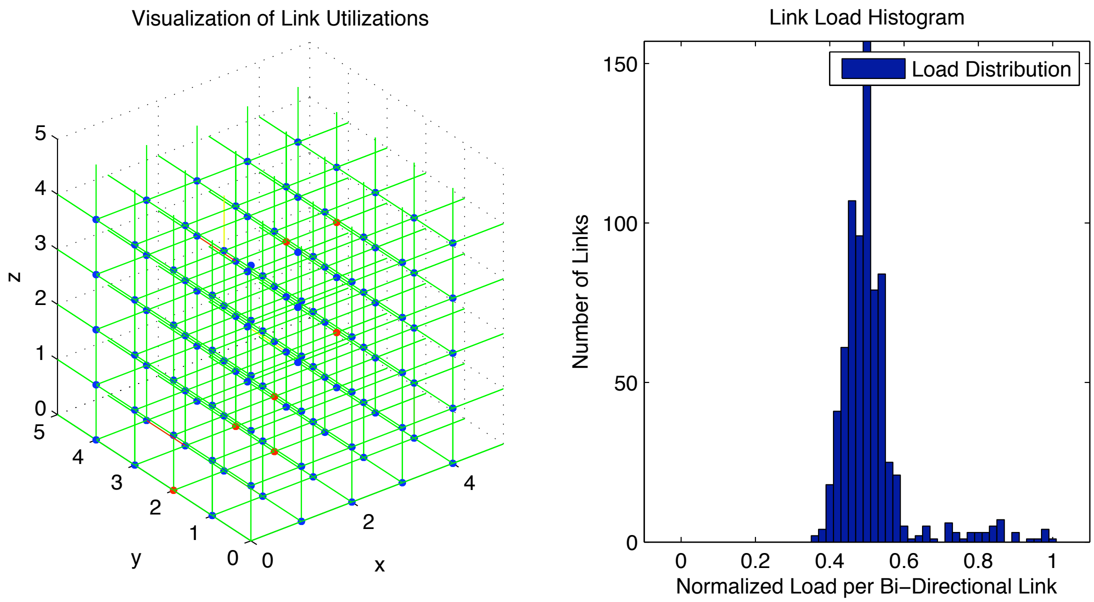


Figure E.82: *Results of CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*

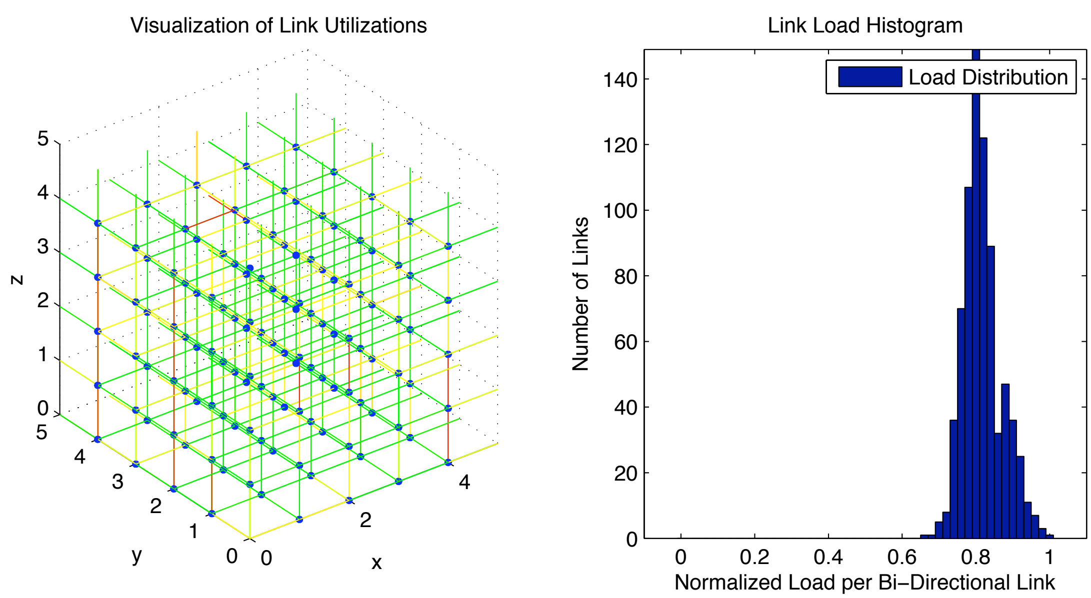


Figure E.83: *Results of Enhanced CQR Routing using the flood traffic demand (FL). Two individual traffic demands were assigned to each node from each node.*

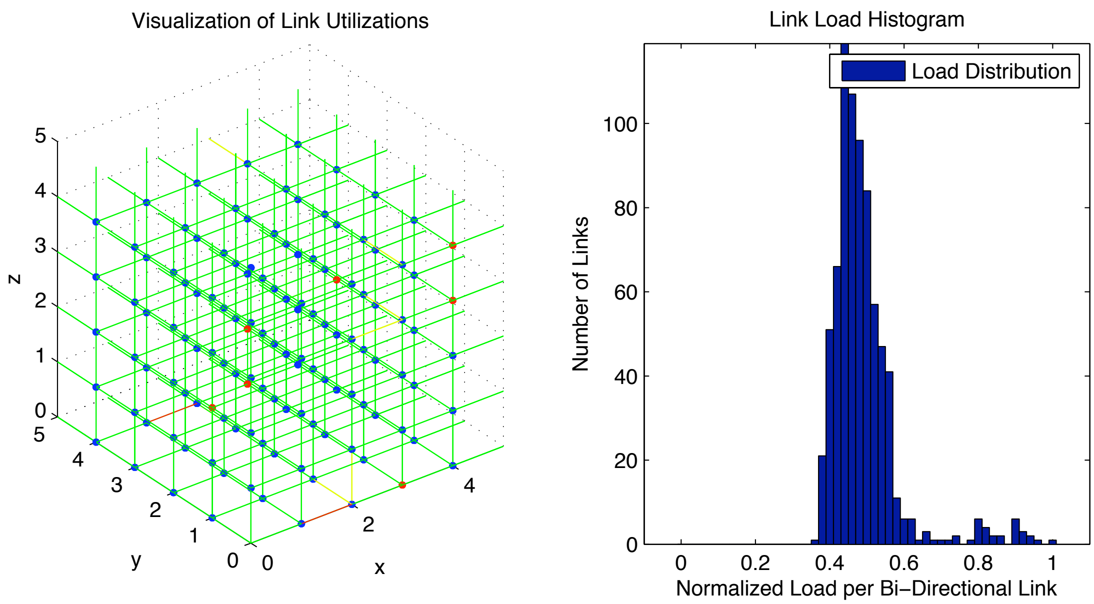


Figure E.84: *Results of Enhanced CQR Routing using the flood traffic demand (FL) with Hot-spots. Two individual traffic demands were assigned to each node from each node. Five percent of the nodes were made hot-spots, and they are marked as red in the figure.*