# From the Wisdom of the Hive to Intelligent Routing in Telecommunication Networks:

## A Step towards Intelligent Network Management through Natural Engineering

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Universität Dortmund

am Fachbereich Informatik

von

**Muddassar Farooq**

Dortmund

2006

Tag der mündlichen Prüfung: 01.02.2006

Dekan: Prof. Dr. Bernhard Steffen

Gutachter: Prof. Dr. Horst F. Wedde (Universität Dortmund)

Prof. Dr. Heiko Krumm (Universität Dortmund)

## Dedication

This thesis is dedicated to my father Barkat Ali and my mother Asmat. This should not be considered as a traditional dedication because my father is not a person but an institution. He retired as a senior bank executive. The financial experts could imagine the stress related to such a job. He used to teach me at least for two to three hours daily in my primary school after coming from his tiring job routine. I still remember that once he was posted in a rural town of Saudi Arabia, I was unable to go to any school for two years because of unavailability of any English or Urdu medium school. However, I have the honor to go to the school of my father. He taught me everything from science to mathematics and from drawing to literature during these two years. I just used to go to Dhahran province at the end of the academic year to take my final examination in an Urdu medium school. Some of you might be surprised to know that I got second position both in grade 5 and grade 6 and nearly missed the top one by a couple of marks. I think that without his tremendous hard work I would have not been able to be successful in my life. I believe that the world would be a better place for many children if their father could give them only 20% of the time that my father gave to me. I thank you and salute you my teacher, tutor and father. This thesis is your thesis and this success is your success. My mother is a house wife and she gave me all what a mother could give to her child. Without her strong encouragement and prayers, I would have not achieved this success in my life. I am thankful to God that He gave me parents like you.

## Acknowledgments

First, I shall thank Prof Dr. Horst F. Wedde, who showed his confidence in me by allowing me to tread on a labyrinthine research path where many other professors would have not even dared to think of. He always encouraged me and remained patient while I was reading the two masterpiece books: *The Dance Language and Orientation of Bees* and *The Wisdom of the Hive*. Finally, his patience and confidence was generously rewarded once our paper won the best paper award at ANTS conference in Brussels in 2004. Currently, we are working on two projects that are inspired from the bee behavior: *BeeHive* deals with routing in fixed networks and *BeeAdHoc* deals with routing in Mobile Ad Hoc Networks (MANETs). The projects have received enormous attention by the Nature inspired routing algorithm groups around the world. Moreover, my special gratitude goes to Prof Wedde, the way he has thoroughly read the draft version of this manuscript. Last but not least he pushed a lazy person like me to the limits to finish the writing of this manuscript in time. I would also like to thank Prof Dr. Heiko Krumm and Dr. Thomas Bartz-Beielstein for their valuable comments and suggestions on an earlier version of the manuscript. These helped in improving the quality of the manuscript.

My stay at LS III is a story of dedicated friendship. I consider this friendship even a bigger achievement than *BeeHive* or *BeeAdHoc*. Frank-Thorsten Breuer and his parents accepted us like family members in their family. Every couple of months they invited us for a dinner or a party at their home. Arnim Wedig took care of me with his nice tea and cookies. He also assisted me in the procurement of expensive computational resources for our projects. Mario Lischka helped me in quickly learning LaTex. I must not dare to forget Mrs Düsenberg, who as we call her, is the heart of our department. She is reputed to be our de facto psychotherapist. She gave me useful tips how to be a successful husband.

*BeeHive* would have never been realized inside the network stack of the Linux kernel without the dedicated work of my students Yue Zhang and Alexander Harsch. I find myself lucky that I had the opportunity to supervise them in their Master theses. Constantin Timm deserves my special indebtedness for developing a plotter utility that automated the process of reading the data files and then plotting the important performance values in the figures. I would also like to thank Gianni Di Caro at IDSIA, Switzerland. We extensively exchanged emails and our discussions resulted in identifying the important directions for our *BeeHive* project.

BeeHive project would have not been successful without two special persons: my wife Saadi (Dua) and my son Yousouf. Saadi is my girl friend, my love and then of course my wife. She has and is still sacrificing her career in order to enable me to quickly finish my projects and the current manuscript. She is a gynecologist and I wish that a day would come when I could do something for her as well. Yousouf kept me busy in everything except my *BeeHive* project. He showed me that there are more important things in life than *BeeHive* e.g. *Teletubbies* and *Barney*. I now remember their names by heart (Tinky-Winky, Dipsy, Laa-Laa and Po) because we have seen them almost daily during past couple of months.

# Contents

**List of Figures** ix

**List of Tables** xi

**List of Algorithms** xii

**1 Introduction** 1
1.1 Motivation of the work . . . 2
1.2 Problem statement . . . 3
1.2.1 Hypotheses . . . 4
1.3 An engineering approach to Nature inspired routing protocols . . . 5
1.4 The scientific contributions of the work . . . 6
1.4.1 A simple, distributed, decentralized multi-agent system . . . 6
1.4.2 A comprehensive routing system . . . 6
1.4.3 An empirical comprehensive performance evaluation framework . . . 6
1.4.4 A scalability framework for (Nature inspired) agent-based routing protocols 8
1.4.5 Protocol engineering of Nature inspired routing protocols . . . 8
1.4.6 A Nature inspired Linux router . . . 8
1.4.7 The protocol validation framework . . . 8
1.5 Organization of the thesis . . . 8

**2 A Comprehensive Survey of Nature Inspired Routing Protocols** 13
2.1 Introduction . . . 13
2.1.1 Organization of chapter . . . 14
2.2 Network routing algorithms . . . 14
2.2.1 Features landscape of a modern routing algorithm . . . 14
2.2.2 Taxonomy of routing algorithms . . . 15
2.3 Ant Colony Optimization (ACO) routing algorithms for fixed networks . . . 19
2.3.1 Important elements of ACO in routing . . . 19
2.3.2 Ant-based control (ABC) for circuit switched networks . . . 20
2.3.3 Ant-based control (ABC) for packet switched networks . . . 22
2.3.4 AntNet . . . 23
2.3.5 Ant Colony Routing (ACR) and AntNet+SELA QoS aware routing . . . 24
2.3.6 A brief history of research in AntNet . . . 25
2.4 Evolutionary routing algorithms for fixed Networks . . . 28
2.4.1 Important elements of EA in routing . . . 29
2.4.2 GARA . . . 29
2.4.3 ASGA and SynthECA . . . 30
2.4.4 DGA . . . 31
2.5 Related work on routing algorithms for fixed networks . . . 34
2.5.1 Artificial Intelligence community . . . 34
2.5.2 Networking community . . . 35
2.6 Summary . . . 39

**3 From *the Wisdom of the Hive* to Routing in Telecommunication Networks** 43
3.1 Introduction . . . 43
3.1.1 Organization of the chapter . . . 44
3.2 An agent-based investigation of a honey bee colony . . . 44
3.2.1 Labor management . . . 44

3.2.2 The communication network of a honey bee colony . . . 45
3.2.3 Reinforcement learning . . . 45
3.2.4 Distributed coordination and planning . . . 45
3.2.5 Energy efficient foraging . . . 45
3.2.6 Stochastic selection of flower sites . . . 46
3.2.7 Group organization . . . 46
3.3 BeeHive: The mapping of concepts from Nature to networks . . . 46
3.4 The *bee agent* model . . . 47
3.4.1 Estimation model of agents . . . 49
3.4.2 Goodness Of a neighbor . . . 51
3.4.3 Communication paradigm of agents . . . 52
3.4.4 Packet switching algorithm . . . 52
3.5 BeeHive algorithm . . . 55
3.6 The performance evaluation framework for Nature inspired routing algorithms . . . 58
3.7 Routing algorithms used for comparison . . . 61
3.7.1 AntNet . . . 61
3.7.2 DGA . . . 61
3.7.3 OSPF . . . 61
3.7.4 Daemon . . . 62
3.8 Simulation environment for BeeHive . . . 62
3.8.1 simpleNet . . . 62
3.8.2 NTTNet . . . 62
3.8.3 Node150 . . . 62
3.9 Discussion of the results from the experiments . . . 64
3.9.1 Congestion control behavior . . . 64
3.9.2 Queue management behavior . . . 80
3.9.3 Hot spots . . . 84
3.9.4 Router crash experiments . . . 84
3.9.5 Bursty traffic generator . . . 90
3.9.6 Session-less network traffic . . . 91
3.9.7 Size of routing table . . . 94
3.10 Summary . . . 96

**4 A Scalability Framework for Nature Inspired Routing Algorithms 99**
4.1 Introduction . . . 99
4.1.1 Existing work on the scalability analysis . . . 100
4.1.2 Organization of the chapter . . . 102
4.2 The scalability model for a routing algorithm . . . 103
4.2.1 Cost model . . . 103
4.2.2 Power model of an algorithm . . . 104
4.2.3 Scalability metric for a routing algorithm . . . 105
4.3 Simulation environment for scalability analysis . . . 105
4.3.1 simpleNet . . . 105
4.3.2 NTTNet . . . 105
4.3.3 Node150 . . . 105
4.3.4 Node350 . . . 106
4.3.5 Node650 . . . 106
4.3.6 Node1050 . . . 106
4.4 Discussion of the results from the experiments . . . 106
4.4.1 Throughput and packet delivery ratio . . . 108
4.4.2 Packet delay . . . 109
4.4.3 Control overhead and suboptimal overhead . . . 112
4.4.4 Agent and packet processing complexity . . . 114
4.4.5 Routing table size . . . 119

4.4.6 Investigation of the behavior of *AntNet* . . . 119
4.5 Towards empirically founded scalability model for routing protocols . . . 121
4.5.1 Scalability matrix and scalability analysis . . . 123
4.5.2 Scalability analysis of BeeHive . . . 124
4.5.3 Scalability analysis of AntNet . . . 125
4.5.4 Scalability analysis of OSPF . . . 125
4.6 Summary . . . 126

**5 BeeHive in real networks of Linux routers 129**
5.1 Introduction . . . 129
5.1.1 Organization of the chapter . . . 130
5.2 Engineering of Nature inspired routing protocols . . . 130
5.2.1 Structural design of a routing framework . . . 131
5.2.2 Structural semantics of the network stack . . . 134
5.2.3 System design issues . . . 136
5.3 Natural routing framework: design and implementation . . . 136
5.3.1 Algorithm-independent framework . . . 137
5.3.2 Algorithmic-dependent BeeHive module . . . 137
5.4 Protocol verification framework . . . 142
5.5 The motivation behind design and structure of experiments . . . 147
5.6 Discussion of the results from the experiments . . . 148
5.6.1 Quantum traffic engineering . . . 148
5.6.2 Real world applications traffic engineering . . . 161
5.6.3 Hybrid traffic engineering . . . 162
5.7 Summary . . . 164

**6 Conclusion and Future Works 165**
6.1 Conclusion . . . 165
6.2 Future research . . . 167
6.2.1 Quality of Service (QoS) routing . . . 167
6.2.2 Cyclic paths . . . 169
6.2.3 Formal analysis framework . . . 169
6.2.4 Security . . . 171
6.2.5 Intelligent and knowledgeable network engineering . . . 171
6.2.6 Bee colony metaheuristic . . . 173
6.3 Natural Engineering: The need for a distinct discipline . . . 174

**A Software Protocol Engineering for Linux Routers 175**
A.1 Networking code . . . 175
A.2 Data structures . . . 176
A.2.1 socket buffer sk_buff . . . 176
A.2.2 Network device structure net_device . . . 177
A.2.3 Socket structure . . . 179
A.3 Datalink layer . . . 180
A.3.1 Receiving packets . . . 180
A.4 Network layer . . . 182
A.4.1 Receiving packets . . . 182
A.4.2 Sending . . . 183
A.5 UDP . . . 183
A.5.1 Data structures . . . 184
A.5.2 Functions . . . 186
A.6 TCP . . . 187
A.6.1 TCP header . . . 187
A.6.2 TCP states . . . 188

A.6.3 Three way handshake . . . . . . . . . . 188
A.7 Routing . . . . . . . . . . 190
A.7.1 Policy routing . . . . . . . . . . 190
A.7.2 Implementation . . . . . . . . . . 190
A.8 NetFilter . . . . . . . . . . 195
A.8.1 Calling hook functions . . . . . . . . . . 195
A.8.2 Searching the hook table . . . . . . . . . . 196
A.8.3 Actions of hook functions . . . . . . . . . . 197
A.9 Nature inspired routing protocols in the Linux kernel . . . . . . . . . . 197
A.9.1 Agent propagation . . . . . . . . . . 197
A.9.2 Queue management . . . . . . . . . . 199
A.9.3 Quality evaluation . . . . . . . . . . 199

**References** **201**

# List of Figures


1.1 Natural protocol engineering . . . 7

2.1 A taxonomy of routing protocols for fixed telecommunication networks . . . 18
2.2 Pheromone routing table in ABC . . . 21
2.3 Routing table in DGA . . . 33
2.4 Routing classification . . . 39

3.1 Bee agents flooding algorithm . . . 50
3.2 Goodness of a neighbor (different options) . . . 53
3.3 Communication paradigm of *bee agents* . . . 54
3.4 Performance evaluation framework . . . 58
3.5 SimpleNet . . . 63
3.6 NTTNet . . . 63
3.7 Node150: figure is captured from OMNeT++ plotter . . . 64
3.8 Congestion control behavior in simpleNet (throughput and packet delay) . . . 65
3.9 Congestion control behavior in simpleNet (packet delivery ratio and session completion ratio) . . . 66
3.10 Congestion control behavior in simpleNet (control and suboptimal overhead) . . . 67
3.11 Congestion control behavior in simpleNet (agent and data processing complexity) . . . 68
3.12 Congestion control behavior in NTTNet (throughput and packet delay) . . . 70
3.13 Congestion control behavior in NTTNet (packet delivery ratio and session completion ratio) . . . 71
3.14 Congestion control behavior in NTTNet (control and suboptimal overhead) . . . 72
3.15 Congestion control behavior in NTTNet (agent and data processing complexity) . . . 73
3.16 Congestion control behavior in Node150 (throughput and packet delay) . . . 75
3.17 Congestion control behavior in Node150 (packet delivery ratio and session completion ratio) . . . 76
3.18 Congestion control behavior in Node150 (control and suboptimal overhead) . . . 77
3.19 Congestion control behavior in Node150 (agent and data processing complexity) . . . 78
3.20 Queue management/control behavior of algorithms (packet delivery ratio and packet delay) . . . 81
3.21 Queue management/control behavior of algorithms (session completion ratio and session delay) . . . 82
3.22 Hot spot is Node 0 in NTTNet . . . 85
3.23 Hot spot is Node 0 in Node150 . . . 86
3.24 Router 20 and Router 43 crashed at time = 500 seconds . . . 88
3.25 Router 20 crashed at 300 seconds and Router 43 crashed at 500 seconds and both were repaired at 800 seconds . . . 89
3.26 Bursty traffic behavior in NTTNet . . . 90
3.27 Bursty traffic behavior in Node150 . . . 92
3.28 Session-less network traffic for NTTNet . . . 93
3.29 Session-less network traffic for Node150 . . . 95
3.30 Size of routing table . . . 96

4.1 The scalability behavior in different situations . . . 101
4.2 Node350: figure is captured from OMNeT++ plotter . . . 106
4.3 Node650: figure is captured from OMNeT++ plotter . . . 107
4.4 Node1050: figure is captured from OMNeT++ plotter . . . 107
4.5 Throughput (Mbits/sec)) . . . 110
4.6 Packet delivery ratio (%) . . . 111

4.7 Packet delay (msec)) . . . . . 113
4.8 Routing overhead (%) . . . . . 115
4.9 Suboptimal overhead (%) . . . . . 116
4.10 Agent processing complexity per node (in billions) . . . . . 117
4.11 Packet switching complexity per node (in billions) . . . . . 118
4.12 The size of the routing table . . . . . 119
4.13 The behavior of AntNet (agents sent, received and deleted) . . . . . 120
4.14 The behavior of AntNet (agent life in hops and sec) . . . . . 121

5.1 Monolithic implementation in kernel space . . . . . 132
5.2 Monolithic implementation in user space . . . . . 133
5.3 Hybrid implementation . . . . . 134
5.4 Protocol block . . . . . 135
5.5 NetFilter hooks . . . . . 138
5.6 Block diagram of BeeHive module . . . . . 139
5.7 Top level routing table . . . . . 141
5.8 IPs table . . . . . 142
5.9 Neighbors table . . . . . 143
5.10 Detailed bee routing table . . . . . 144
5.11 Protocol verification framework . . . . . 145
5.12 simpleNet topology: 8 routers, 9 bidirectional links each of 10 Mbits/sec . . . . . 146
5.13 Experiments 1-4 (packet delivery ratio and packet delay) . . . . . 150
5.14 Experiments 1-4 (session completion ratio and session delay) . . . . . 151
5.15 Experiments 5-7 (packet delivery ratio and packet delay) . . . . . 153
5.16 Experiments 5-7 (session completion ratio and session delay) . . . . . 154
5.17 Experiment 8 (throughput and packet delay) . . . . . 155
5.18 Hot spot experiments (packet delivery ratio and packet delay) . . . . . 157
5.19 Hot spot experiments (session completion ratio and session delay) . . . . . 158
5.20 Router down (packet delivery ratio and packet delay) . . . . . 159
5.21 Router down (session completion ratio and session delay) . . . . . 160
5.22 FTP experiments . . . . . 161
5.23 FTP experiments with 15 downloads . . . . . 162

6.1 Routing classification . . . . . 167
6.2 Jitter (msec) . . . . . 168
6.3 Distribution of packets that follow cyclic paths . . . . . 170
6.4 Nature inspired distributed and autonomous router . . . . . 172

A.1 Networking code in the Linux kernel tree . . . . . 175
A.2 Software architecture . . . . . 181
A.3 TCP packet processing . . . . . 189
A.4 fib_table structure . . . . . 193
A.5 fib_node structure . . . . . 194

# List of Tables


2.1 Wired best-effort networks . . . . . . . . . . 28
2.2 Wired QoS networks . . . . . . . . . . 28
2.3 Classification of routing algorithms for fixed networks . . . . . . . . . . 40

3.1 Processing complexity of different forms . . . . . . . . . . 52
3.2 *Intra foraging zone* routing table . . . . . . . . . . 55
3.3 Symbols used in the *BeeHive* algorithm . . . . . . . . . . 57
3.4 Input parameter symbols used in the chapter . . . . . . . . . . 59
3.5 Output parameter symbols used in the chapter . . . . . . . . . . 60
3.6 Performance parameters for congestion control behavior in simpleNet . . . . . . . . 69
3.7 Performance parameters for congestion control behavior experiments for NTTNet . 74
3.8 Performance parameters for congestion control behavior in Node150 . . . . . . . . 76
3.9 Performance parameters for different buffer capacities in NTTNet . . . . . . . . . . 83
3.10 Performance parameters for hot spot and router down experiments . . . . . . . . . 87
3.11 Performance parameters for bursty traffic generators on NTTNet . . . . . . . . . . 91
3.12 Performance parameters for bursty traffic generators on Node150 . . . . . . . . . . 91
3.13 Performance parameters for Session-less traffic in NTTNet . . . . . . . . . . 94
3.14 Performance parameters for Session-less traffic in Node150 . . . . . . . . . . 95

4.1 Performance values for MSIA = 4.6 sec . . . . . . . . . . 109
4.2 Performance values for MSIA = 2.6 sec . . . . . . . . . . 109
4.3 Performance values for MSIA = 1.6 sec . . . . . . . . . . 112
4.4 BeeHive . . . . . . . . . . 122
4.5 AntNet . . . . . . . . . . 122
4.6 OSPF . . . . . . . . . . 123
4.7 Scalability Matrix for BeeHive . . . . . . . . . . 125
4.8 Scalability Matrix for AntNet . . . . . . . . . . 126
4.9 Scalability Matrix for OSPF . . . . . . . . . . 126

5.1 Symbols used in the chapter . . . . . . . . . . 145
5.2 The mapping of Hosts to IP Addresses in SimpleNet . . . . . . . . . . 147
5.3 Parameters for traffic generator for Experiments 1 to 10 . . . . . . . . . . 148
5.4 Important performance values for Experiments 1 to 4 from . . . . . . . . . . 152
5.5 Important performance values for Experiments 5 to 7 . . . . . . . . . . 152
5.6 Important performance values for Experiment 8 . . . . . . . . . . 156
5.7 Important performance values for hot spot experiments . . . . . . . . . . 156
5.8 Important performance values for router down experiments . . . . . . . . . . 160
5.9 Performance values from SQTG and D-ITG . . . . . . . . . . 163
5.10 Important performance values for UDP and VoIP experiments . . . . . . . . . . 164

# List of Algorithms

1 Bee launching and processing algorithms . . . . . . . . . . . . . . . . . . . . . . . . 56
2 Packet switching and neighbor maintenance algorithms . . . . . . . . . . . . . . . . . 57

# 1

# Introduction

During the past years, telecommunication networks have become a special focus of research, both in academia and industry [76, 77, 162]. This is certainly due to the unprecedented growth of the Internet during the last decade of the previous century as it developed into a nerve center of the communication infra-structure [136]. One important reason for the success of the Internet is its connection-less packet-switching technology (no connection is established between a sender and a receiver). Such a paradigm results in a simple, flexible, scalable and robust network layer architecture [109, 15, 152]. This is in contrast to traditional connection-oriented telecommunication networks in which a circuit is reserved for a connection between a sender and a receiver [76, 77, 162].

The Internet success motivated researchers to realize the dream of *Ubiquitous Computing*, including the concept of "one person–many computers" [223, 225, 224, 226]. Research and development in *Ubiquitous Computing* resulted in an exponential growth of smart hand-held computing devices, which have to be inter-connected and connected to the Internet to satisfy highly demanding users. In turn, these requirements resulted in a phenomenal growth in wireless telecommunication networks and their supporting Internet Protocol (IP) (which is the standard protocol for the network layer of the Internet) on wireless networks. However, these wireless networks require an infra-structure (base station) for providing connectivity to mobile terminals. In the sequel, work on Mobile Ad-Hoc Networks (MANETs) has become a vigorous effort. Here mobile terminals communicate with one another without the need for a communication infra-structure. These networks have turned useful or even indispensable in search and rescue operations, disaster relief management and military command and control.

*Ubiquitous Computing* has created a demanding community of users, who are utilizing its potential in novel applications like World Wide Web (WWW), Computer Supported Collaborative Work (CSCW) Environments, E-commerce, Tele-medicine, E-learning etc. An essential feature of most of these applications is the ability to transmit audio and video streams to the participants under some Quality of Service (QoS) constraints. The users want all of these services on their desktops as well as on their mobile terminals. Such challenging requirements can only be met if a network's resources are utilized in an efficient manner.

The efficient utilization of limited network resources and infra-structures by enhancing/optimizing the performance of operational IP networks is defined as *Traffic Engineering* [12, 135]. These goals are accomplished by devising efficient and reliable routing strategies. The important features and characterizations of such routing protocols are: *load-balancing, constraint-based routing, multipath-routing, fast re-routing, protection switching, fault-tolerance and intelligent route management*. Currently, the Internet community employs multi-path routing algorithms like MPLS (Multi-protocol Label Switching) [148], which is based on managing virtual circuits on top of the IP layer, and hence lacks scalability and robustness. Another approach avoids completely the use of virtual circuits and manages the resources of each session by doing per-flow fair scheduling of

the links. Nevertheless, flows are setup along the shortest paths determined by the underlying routing protocols. The reservation of flows are managed by Resource Reservation Protocol (RSVP) [243, 207]. However, the deterministic service guarantees are provided to real time applications using the Interserv architecture [24, 243]. In large networks, this per flow mechanism does not scale (they have hundreds of thousands of flows), therefore, RSVP has been extended in [86] by replacing the per-flow routing state with per source/destination routing state. This results in a state size that grows only quadratically with the number of nodes. Both of these protocols suffer from serious performance bottlenecks because they utilize the single-path routing algorithm *Open Shortest Path First (OSPF)* at IP layer. Consequently, the bandwidth of the single path is quickly consumed which results in a high call blocking probability [207].
The major challenge in *traffic engineering* in a nutshell is to *to design multi-path routing protocols for IP networks in which multiple/alternative paths are efficiently discovered and maintained between source and destination pairs*. Such routing protocols will provide solutions to existing technical challenges, by using the connection-less paradigm of the IP layer.

## 1.1 Motivation of the work

We believe that a complete reengineering of the network layer is the logical solution to not only *traffic engineering* problem but also to network management. The growth of the Internet demands design and development of novel and intelligent routing protocols that would result in an intelligent and knowledgeable network layer. Currently, the network layer is relegated to just switching data packets to the next hop based on the information in the routing tables collected by non-intelligent control packets. The new protocols, however, have to be designed with a careful engineering vision in order to reduce their communication, processing and router's resource costs.
The research in agent-based routing systems has resulted in developing many novel networking systems [199, 44, 89, 132]. The algorithms utilize software agents, which have the following properties [247]:

- *Autonomous:* the capability of performing autonomous actions.
- *Proactive:* the capability of exhibiting opportunistic and goal-oriented behavior and taking initiative where appropriate.
- *Responsiveness:* the capability of perceiving the environment and responding in a timely fashion to the changes that occur in it.
- *Social:* the capability of interaction with other artificial agents and humans when appropriate in order to achieve their own objectives and to help others in their activities.

This design paradigm, therefore, focuses on robust and intelligent agent behavior. In [228], White blames Artificial Intelligence (AI) community for this state of the helm. The AI community has been strongly influenced by *Symbol Hypothesis* [143] and first order predicate logic. The symbols and theorem proving are the classical tools, which are based on *Resolution Principle* [158]. Consequently, such systems coordinate their activities by exchanging symbolic information and theorem proving. In addition, all properties of a system could not be inferred by representing knowledge in a symbol formula and then manipulating it using the first order predicate logic [161, 228]. Another shortcoming is the *Frame* problem, which results due to the need of specifying state and state transitions. The measured data obtained from real world systems has to be represented in symbols, which leads to the *sensor fusion* problem. The connectionist systems or artificial neural networks try to overcome these problems. However, their black box nature makes it difficult to synthesize and utilize them in distributed network systems [228].
The real world networks represent a dynamic environment in which good routing decisions need to be taken in real time under a number of performance and cost constraints, therefore, applying such complex paradigms to achieve intelligence in the network layer is not feasible. The processing

complexity and communication cost of launching such complex agents will be overwhelming and they would also consume significant amounts of a router's resources, especially in large networks. The above-mentioned problems in traditional agent-based approaches could be easily solved if we follow a dramatically novel paradigm for designing the agents: *agents need not be rational in order to solve complex problems* [228]. This conjecture, at first, appears to completely boggle the mind because it suggests that *intelligence could result from simple non-intelligent agents*. However, the systems which are based on this design paradigm are rigorously studied in Swarm Intelligence [17]. It takes the inspiration from self-organization in natural colony systems e.g. ants or bees [27] and utilizes their principles as a metaphor to design simple agents that take decisions based on local information without the need of a central complex controller. However, such agents are situated in their environment and they utilize either a direct agent-agent communication paradigm or an agent-group paradigm in which they indirectly communicate through the environment. In [27, 20], the authors have defined the basic ingredients of Self-organization, which are the following:

1. The *positive feedback* in the system amplifies the good solutions that the agents have discovered. Consequently, other agents are recruited to exploit these good solutions.
2. The *negative feedback* in the system helps in counterbalancing the positive feedback, as a result, good solutions could not dominate forever.
3. *Amplification of random solutions* helps in discovering and exploring new solutions.
4. *Multiple interactions* help in enabling individuals to use the result of their own activities as well as of others' activities.

In this way a colony is able to achieve a complex and intelligent behavior at a colony level which is well beyond the intelligence and capabilities of an individual in the colony. We believe that self-organization systems have all the features that we could wish in large network systems.

## 1.2 Problem statement

We believe that the complexity of the manifold task of endowing intelligence and knowledge to network layer through self-organizing agents, which are inspired from communicative and evaluative principles of a honey bee colony, is overwhelmingly phenomenal. Therefore, in our research, we take a cardinal first step to achieve this objective. Our problem statement could be outlined as: *efficient, scalable, robust, fault-tolerant, dynamic, decentralized and distributed solutions to the traffic engineering could be provided within the existing connection-less model of IP through a Nature inspired population of agents, which have simple behavior. The agents explore multiple paths between all source/destination pairs and then distribute the network traffic on them. This approach could significantly enhance the network performance.*
Our routing protocol should be able to meet the following challenging requirements:

1. The *agents* must not require existing Multi-Agent System (MAS) software for their realization. Rather their behavior and learning algorithm should be simple enough to be implemented directly in the network layer by utilizing semantics of C/C++ languages.
2. The processing complexity of *agents* must be kept at a minimum level and the time a router spends in processing them should only be a fraction of the time that it spends in switching data packets. This requirement is necessary because the performance of a router could significantly degrade if *agent processing* steals mosts of its time [242].
3. The *agents* must explore the network in an asynchronous manner.
4. It must be robust to loss of *agents*.
5. The size of agents must be such that they could fit into the payload of an IP packet. This requirement will significantly reduce the communication related costs.

6. It must be able to scale to large networks.
7. It must be designed with a vision to install it on real world routers. Therefore, the simulation model must be realizable inside the network stack of a Linux router.
8. It must be realizable in real world routers without the need for additional resources both in hardware or software. This requirement would simplify its installation, though in a cost effective manner, on existing routers.
9. It must not require synchronization of clocks in the network.
10. It must not require that the routing tables of different routers should be in a consistent state for taking correct routing decisions.

### 1.2.1 Hypotheses

The study of honey bees has revealed a remarkable sophistication of their communication capabilities. Nobel laureate Karl von Frisch deciphered and structured these into a language, in his book *The Dance Language and Orientation of Bees* [206]. Upon their return from a foraging trip, bees communicate the distance, direction, and quality of a flower site to their fellow foragers by making waggle dances on a dance floor inside the hive. By dancing zealously for a good foraging site they recruit foragers for the site. In this way a good flower site is exploited, and the number of foragers at this site are reinforced. A honey bee colony has many features that are desirable in networks:

- efficient allocation of foraging force to multiple food sources;
- different types of foragers for each commodity;
- foragers evaluate the quality of food sources visited and then recruit optimum number of foragers for their food source by dancing on a dance floor inside the hive;
- no central control;
- foragers try to optimize the energetic efficiency of nectar collection and foragers take decisions without any global knowledge of the environment.

In our work we make the following hypotheses

(a) **H1:** If a honey bee colony is able to adapt to countless changes inside the hive or outside in the environment through simple individuals without any central control, then an agent system based on similar principles should be able to adapt itself to an ever changing network environment in a decentralized fashion with the help of simple agents who rely only on local information. This system should be dynamic, simple but efficient, robust, flexible, reliable and scalable because its natural counterpart has got all of these features.

(b) **H2:** If designed with a careful engineering vision, Nature inspired solutions are simple enough to be installed on real world systems. Therefore, their benefit-to-cost ratio should be better as compared with existing real world solutions.

We believe that all of these objectives can be achieved by contemplating on novel paradigms for developing agents. The research, however, is of multidisciplinary nature because it involves cross-fertilization of ideas from Biology, AI, Agent Technology, Network Management and Network Engineering etc. Therefore, we developed a *Natural Engineering* approach [1] to successfully accomplish our objectives in a given time frame.

[1] The focus of our work is on following an engineering approach for Nature inspired routing protocols. However, the engineering approach itself is general enough and complements the existing approaches of Bionik [142, 160] and CI (Computational Intelligence) [3].

## 1.3 An engineering approach to Nature inspired routing protocols

In this section we will introduce our engineering approach that we followed in the design and development of a routing protocol inspired from a natural system (a honey bee colony).

**Definition 1 (Natural Engineering)** *Natural Engineering is an emerging engineering discipline that enables scientists/engineers to utilize inspirations and observations from organizational principles of natural systems, and to transform them into structural principles of software organization of algorithms or industrial product design, in search of efficient/optimal solutions for real world problems under resource constraints.*

The above-mentioned concept emphasizes six aspects:

1. Understanding the working principles of natural systems.
2. Developing algorithmic models of the organizational principles of natural systems.
3. Understanding the operational environment of target systems.
4. Mapping the concepts from the natural system to the technical system.
5. Adapting the algorithmic model to the operational environment of a technical system.
6. Following a testing and evaluating feedback loop in search of optimum solutions under the resource constraints (time, space, computation, money, labor etc.).

There is no clear-cut way to achieve a 1-1 match between structures/principles in Nature life organizations and working principles in technical systems. The most important challenge, therefore, is to identify a natural system of which the working principles could be appropriately abstracted, for deriving suitable principles to work in a given technical system. Instead of adding numerous non-biological features to a natural system, we believe that it is more advisable to look to other natural systems for inspiration. In our case we chose honey bee colonies because the foraging behavior of bees could be transformed into different types of agents performing different routing tasks in telecommunication networks. Both systems have to maximize the amount of a commodity (nectar delivered to hives and data delivered to nodes respectively) as quickly as possible, under a permanently and even unpredictably changing operating environment.
The major focus of research is to design and develop cost efficient Bio/Nature inspired business solutions for highly competitive markets. Therefore, the development of a Nature inspired routing algorithm must follow a feedback oriented engineering approach (see Figure 1.1) that incorporates most of the features discussed above.
First we considered the ensemble of constraints under which the envisioned routing protocol is supposed to operate:

- Non-availability of a global clock for trip time calculation.
- Routers and links could crash.
- Routers have limited queue capacity.
- Links have a BER (bit error rate) associated with them.
- The requirements from the Linux kernel routing framework needed to support the protocol.
- The requirements of the IP protocol, which is currently used in the network layer in the Internet.

At the same time we decided that the *bee agents* should explore the network, collect important parameters, and make the routing decisions in a decentralized fashion (in the style as real scouts/foragers do decision making during collecting nectar from flowers). *Bee agents* should measure the quality of a route and then communicate it to other *bee agents* like foragers do in Nature. The structure of the routing tables should provide the functionality of a dance floor for exchanging information among *bee agents* as well as among *bee agents* and data packets. Moreover, we must be able to realize it in a real kernel of the Linux operating system later on.
We implemented our ideas in a simulation environment and then refined our algorithmic mapping through the feedback channel 1 (see Figure 1.1). During this phase we did not use any simulation specific features that were not available inside the Linux kernel, e.g. vector, stack or similar data structures. Once we reached a relative optimum of the *BeeHive* concept, we started to develop an engineering model of the algorithm. The engineering model can be easily transported to the Linux kernel routing framework. We tested it in the real network of Linux routers and refined our engineering model through the feedback channel 2 (see Figure 1.1). We evaluated our conceptual approach in two prototype projects: *BeeHive* [221], which deals with the design and development of a routing algorithm for fixed networks, and *BeeAdHoc*, the goal of which is to design and develop an energy efficient routing algorithm for Mobile Ad Hoc Networks (MANETs) [217, 218, 219].

## 1.4 The scientific contributions of the work

In this section we will list the general scientific contributions achieved during our research in the past four years. The reader will appreciate the overwhelming complexity of the work due to the diverse nature of accomplishments achieved in the *BeeHive* project. Some of the information might be duplicated here but we believe that it is important to make the section self-contained.

### 1.4.1 A simple, distributed, decentralized multi-agent system

We have developed a simple and distributed multi-agent system in which a population of agents collectively achieve an objective. The agents are simple entities with limited processing and memory capabilities and they take their decisions based on their local view of the network state. The state is determined by local information, which is collected in a small region around their launching node. Such a simple agent model is the result of borrowing communication principles from the wisdom of the hive. The agents try to undertake the daunting task of optimizing a number of competing performance values like throughput, packet delay etc. under different cost constraints.

### 1.4.2 A comprehensive routing system

The multi-agent system, as described above, was instrumental in designing and developing a multi-path routing protocol, *BeeHive*, which is dynamic, simple, efficient, robust, flexible and scalable. As demonstrated by our results, the algorithm achieves a similar or better performance as compared with the existing state-of-the-art algorithms. *BeeHive*, however, achieves this objective with significantly lesser costs in terms of processing, communication and router's resources. The algorithm does not require an access to the complete network topology rather it works with a local view of the network. The agents take their decisions in an autonomous and decentralized fashion.

### 1.4.3 An empirical comprehensive performance evaluation framework

The other major contribution of the work is a comprehensive performance evaluation framework, which calculates a number of important performance values and the associated costs of a routing algorithm. The framework can also vary a number of network configurations from traffic patterns to network topology. As a result, the developer of a routing protocol can study the behavior of an

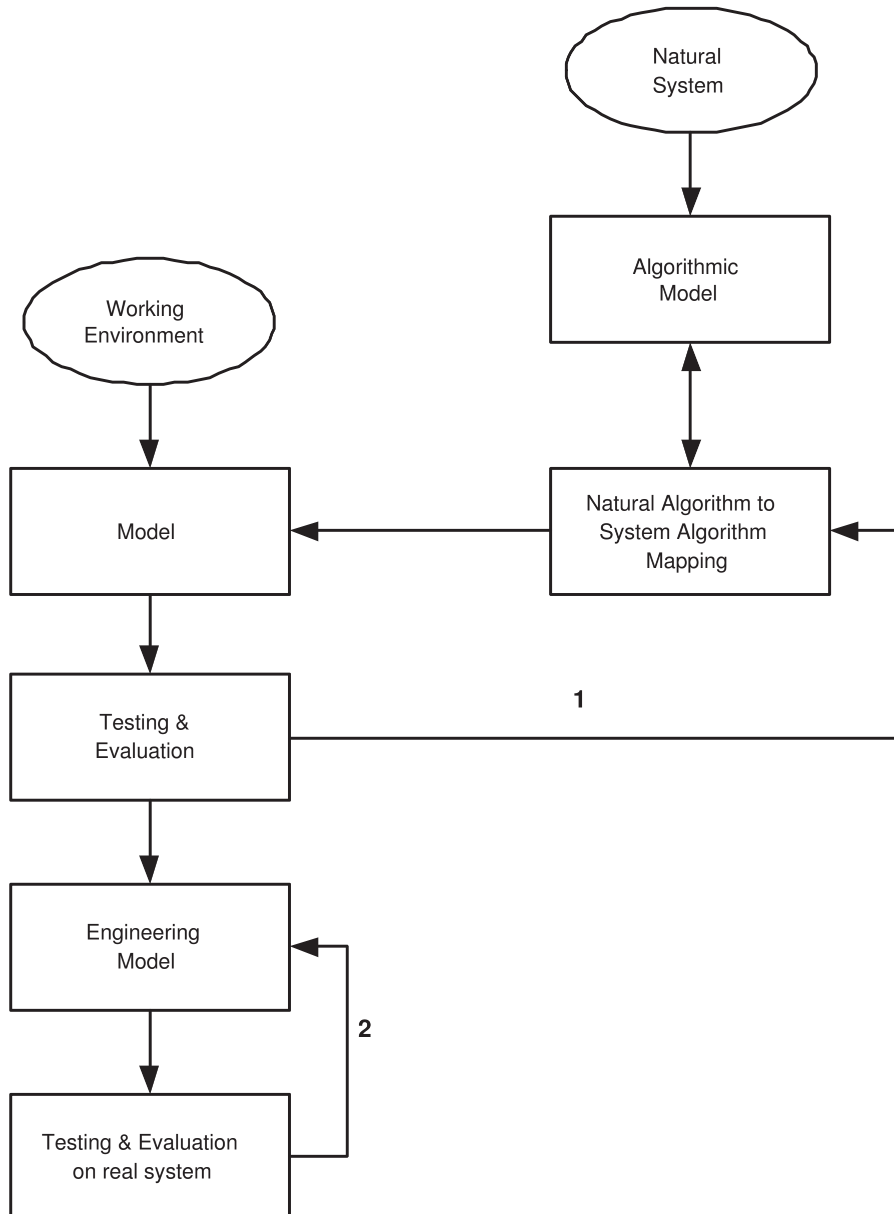



Figure 1.1: Natural protocol engineering

algorithm on a wide operational landscape with a focus on its benefit-to-cost ratio in an unbiased manner. The framework proved to be useful in identifying reasons behind the anomalous behavior of *BeeHive* in different scenarios. Subsequently, we were able to improve our algorithm through the feedback channel 1 as shown in Figure 1.1.

### 1.4.4 A scalability framework for (Nature inspired) agent-based routing protocols

We developed a comprehensive framework that facilitates the study of the scalability of agent-based distributed systems in general and of routing protocols in particular. The framework provides a formal model and a set of empirical tools to protocol developers that are useful in investigating the scalability of their protocols at an early stage of development. To our knowledge, this is the first model that provides an unbiased way of studying the scalability of (Nature inspired) agent-based routing protocols.

### 1.4.5 Protocol engineering of Nature inspired routing protocols

One of the most important contributions of our work is the vision of *Natural Engineering* which has been introduced in the last section. We believe that developing a Nature inspired system, which can be installed or utilized in real world systems is a challenging task. The Nature inspired community, at times, lack the vision about the real operational environments. As a result, most of the proposed solutions were never realized in the intended real world systems. Our work, according to our knowledge, is an important step from "Swarm Intelligence" to "Natural Engineering". We believe that the work will stimulate other researchers to adopt a similar approach for their projects as well.

### 1.4.6 A Nature inspired Linux router

Our *Natural Engineering* approach significantly helped us in developing an algorithmic model in the simulation environment that is mostly independent of the underlaying features of a simulation system. It rather utilizes only those components in a simulation environment which are available in real world Linux routers. This approach showed its benefits once we started developing an engineering model in the form of a Nature inspired Linux router because we were able to make this quantum leap with significantly limited man power and computing resources.

### 1.4.7 The protocol validation framework

Another important contribution of the work is a comprehensive validation framework in which we implemented the same traffic generators in the simulation and in an application layer of a Linux network stack. We also utilized the same network topology both in simulation and real network of Linux routers. Our validation principle is: if we generate the same traffic patterns in identical topologies both in simulation and real network, then the performance values of the algorithms should be traceable from one environment to another with acceptable deviations. We are happy to report that, according to our knowledge, *BeeHive* is the first Nature inspired algorithm which has been implemented in real networks and has shown substantial performance benefits for existing real world applications.

## 1.5 Organization of the thesis

The work presented in this thesis is organized into six chapters. Each chapter, except the first and the last, will provide a comprehensive review of the research conducted in a particular phase of our *Natural Engineering* cycle, which starts from conceiving the ideas from the working principles of a natural system, developing an algorithmic model from them, and realizing the algorithmic model

both in a simulation environment and in a real network of Linux routers. The realization phase, both in simulation and real networks, is complemented by extensive testing, analysis, evaluation and feedback channels.

**Chapter 2–A Comprehensive Survey of Nature Inspired Routing Protocols.**
The chapter presents the true challenges that a routing protocol is expected to meet in complex networks of the new millennium. We provide classifications of the algorithms either based on their characteristics or on their design philosophy. The basic objective of the survey is to understand the design doctrine of different communities that are involved in the design and development of routing algorithms. This will motivate the researchers to develop state-of-the-art routing algorithms through a process of cross-fertilization of useful features and characteristics of different design doctrines. We classify the communities into three categories: Networking community, Artificial Intelligence (AI) community and Natural Computing (NC) community. The focus of the survey presented in Chapter 2 is on the algorithms developed by the Natural Computing community. We provide a detailed survey of routing algorithms which are inspired from the pheromone laying principles of ant colonies. The algorithms are based on Ant Colony Optimization (ACO) metaheuristic. We also provide a comprehensive review of the routing algorithms which are based on the principle of evolution in natural systems. Later in the chapter, we introduce the routing algorithms which are based on the principles of *Reinforcement Learning*. These routing algorithms are developed by Artificial Intelligence community. Finally, we briefly summarize the routing algorithms, which have recently been developed by the Networking community. The comprehensive survey proved helpful in identifying the merits and deficiencies of existing state-of-the-art routing protocols developed by different communities. The chapter is based on the following technical report.

- Horst F. Wedde and Muddassar Farooq. *Nature Inspired Routing Algorithms for Telecommunication Networks: A Comprehensive Survey.* Technical Report 802, Department of Computer Science, University of Dortmund (a part of this report will appear in a special issue of Elsevier Journal of System Architecture on Nature Inspired Applied Systems (NIAS) in the summer of 2006).

**Chapter 3–From *the Wisdom of the Hive* to Routing in Telecommunication Networks.**
The chapter describes the most important steps in our *Natural Engineering* approach. The chapter starts with a brief introduction to the foraging principles of a honey bee colony. We present the biological concepts in such a manner that the reader conveniently conceives a honey bee colony as a population based multi-agent system, in which simple agents coordinate their activities to solve the complex problem of the allocation of labor to multiple forage sites in dynamic environments. The agents achieve this objective in a decentralized fashion with the help of local information that they acquire while foraging. We argue that an efficient, reliable, adaptive and fault-tolerant routing algorithm has to also deal with similar daunting issues.
We then provide the mapping of concepts from a natural honey bee colony to an artificial multi-agent system, which can be utilized for routing in telecommunication networks. The mapping of concepts appears to be a crucial step in developing an algorithmic model of an agent-based routing system. We emphasize the motivation behind important design principles of our *BeeHive* routing algorithm. We provide a comprehensive description of our *bee agent* model by emphasizing the communication paradigm utilized by the *bee agents*, which is instrumental in reducing the costs associated with a routing algorithm: communication, processing and router's resources. Later in the chapter, we introduce our comprehensive empirical performance evaluation framework that calculates a number of preliminary and auxiliary performance values. These values provide an in depth insight into the behavior of a routing algorithm under a variety of challenging network configurations.
Finally, we introduce our extensive experimental framework in a simulation environment. The experiments are designed through extensive brainstorming exercises in order to meticulously analyze

the behavior of a routing protocol under diversified network operations. The results obtained from our performance evaluation framework are discussed. We compare *BeeHive* with a state-of-the-art ACO routing algorithm, *AntNet*, a state-of-the-art evolutionary routing algorithm *Distributed Genetic Algorithm (DGA)*, *OSPF* and *Daemon*. *Daemon* is an ideal algorithm that can instantly access the complete network topology and size of the queues in all routers to take an optimum routing decision. The algorithm, though, is not realizable in real networks due to the associated costs, but, nevertheless, serves as an important benchmark for different algorithms.
The results of the experiments unequivocally suggest that *BeeHive* is able to achieve similar or better performance under congested loads as compared with *AntNet* and is able to achieve similar or better performance under normal static loads as compared with *OSPF*. However, this excellent performance of *BeeHive* is achieved with significantly smaller communication and processing costs and the routing tables which have the order of the size as in *OSPF*. The chapter is based on the following published papers [221, 216], and the technical report [215]:

1. Horst F. Wedde, Muddassar Farooq, and Yue Zhang. BeeHive: An Efficient Fault Tolerant Routing Algorithm under High Loads Inspired by Honey Bee Behavior. In Marco Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, and T. Sttzle, editors, Proceedings of the Fourth International Workshop on Ant Colony and Swarm Intelligence (ANTS 2004), volume 3172 of Lecture Notes in Computer Science, pages 83-94, Brussels, Belgium, September 2004. Springer Verlag. **Winner of the Best Paper Award ANTS 2004**.

2. Horst F. Wedde and Muddassar Farooq. A Performance Evaluation Framework for Nature Inspired Routing Algorithms. In Franz Rothlauf and other, editors, Applications of Evolutionary Computing – Proceedings of EvoWorkshops 2005, volume 3449 of Lecture Notes in Computer Science, pages 136-146, Lausanne, Switzerland, March/April 2005.

3. Horst F. Wedde and Muddassar Farooq. *BeeHive: Routing Algorithms Inspired by Honey Bee Behavior*. K"unstliche Intelligenz. Schwerpunkt: Swarm Intelligence, 18–24, Nov 2005.

4. Horst F. Wedde and Muddassar Farooq. *BeeHive: New Ideas for Developing Routing Algorithms Inspired by Honey Bee Behavior* In Handbook of Bioinspired Algorithms and Applications, Albert Zomaya and Stephan Olariu, Ed. Chapman & Hall/CRC Computer and Information Science, Chapter 21, 321–339, 2005.

5. Horst F. Wedde and Muddassar Farooq. BeeHive: An Efficient, Scalable, Adaptive, Fault-tolerant and Dynamic Routing Algorithm Inspired from the Wisdom of the Hive. Technical Report 801, Department of Computer Science, University of Dortmund.

**Chapter 4–A Scalability Framework for Nature Inspired Routing Algorithms.**
The chapter presents a new scalability framework that designers and developers of the routing algorithms, in general, and of Nature inspired routing protocols, in particular, can utilize to analyze the scalability of their routing protocols. We believe that our new framework will enable the designers of routing protocols to establish the scalability of their routing protocol in an early stage of *protocol engineering* [113]. Such a framework will be instrumental in practicing the principles of *Software Performance Engineering (SPE)*, which also emphasizes the consideration of performance and scalability issues early in the design and architectural phase in order to rectify the deficiencies in a simulation environment. This will not only obviate the risk of a disaster once the algorithm is deployed on large scale networks, but also avert the cost overruns due to tuning or redesign of the algorithm later in the protocol engineering cycle. Consequently, such a pragmatic protocol engineering cycle will be capable of reducing the time to market of a new protocol.
Our scalability model defines power and productivity metrics for a routing protocol. The productivity metric provides an insight into the benefit-to-cost ratio of a routing protocol. The cost model includes the communication, processing and memory costs related to a routing algorithm. We believe that the productivity of a routing algorithm is an important performance value which can be used for an unbiased investigation of a routing protocol. Later we define a scalability

metric which is a ratio of productivity values of two network configurations and its value should be ideally 1 if the algorithm is perfectly scalable from one network configuration to the other.
The framework is general enough to act as a guideline for analyzing the scalability of any agent-based network system. However, in our work, we restricted our analysis to only three protocols due to lack of high performance simulation platforms. We studied the scalability behavior of *BeeHive*, *AntNet* and *OSPF* in 6 topologies which vary in their degree of complexity and connectivity. The size of the topologies is gradually increased from 8 nodes to 1050 nodes. According to our knowledge, this is the first extensive effort to empirically study the scalability of Nature inspired routing protocols.
The results demonstrate that *BeeHive* is able to deliver superior performance both under high or low network traffic loads in all topologies. We believe that an engineering vision during the design and development phase, in which we emphasized the scalability as an important metric, has significantly helped in achieving better scalability metrics for the majority of the network configurations as compared with *AntNet* and *OSPF*. It took more than six months to extensively evaluate the algorithms under a variety of network configurations. The chapter is based on the following technical report.

- Horst F. Wedde and Muddassar Farooq. *A Scalability Framework for Agent Based Routing Protocols*. Technical Report 804, Department of Computer Science, University of Dortmund.

**Chapter 5–*BeeHive* in real networks of Linux routers.**
This chapter describes the second phase of our *Natural Engineering* approach: the realization of an engineering model of *BeeHive* inside the network stack of the Linux kernel and then comparing its performance values with *OSPF* in a real network of eight Linux routers. The work presented in the chapter is novel in the sense that, to our knowledge, *BeeHive* is the first Nature inspired routing algorithm which has been realized and tested in real networks.
The chapter begins by illustrating different design options that are available for realizing a Nature inspired routing algorithm in a Linux router. We then describe the motivation behind our *engineering model* that we realized in a Linux router. Subsequently, we define the software architecture of our Nature inspired Linux router. Here, we emphasize the challenges that we encountered because of the unique features of the *BeeHive* algorithm.
We also migrated our *performance evaluation framework* to the application level of the Linux network stack. The motivation behind this significant step is to follow the protocol verification principle: *if we generate the same traffic patterns through the same traffic generators both in simulation and real networks and utilize the same performance evaluation framework again both in simulation and real networks then the performance values obtained from the simulation environment should be traceable to the ones obtained from real Linux network with minor deviations provided our simulation environment depicts a somewhat realistic picture of a real network.* We believe that this verification principle will help in tracking the performance values in simulation with their counterparts in real networks. If the values were similar, then this would strengthen our thesis: *Nature inspired routing protocols, if engineered properly, could manifest their merits in real networks.*
Finally, we discuss the results obtained from extensive experiments both in simulation and in a real network. We feel satisfied because the performance values obtained from the simulation are consistent to the values in the real network with an acceptable degree of deviation. This, according to our knowledge, is the first substantive work which shows the benefits of utilizing Nature inspired routing protocols in real networks running real world applications, e.g. File Transfer Protocol (FTP) and Voice over IP (VoIP). The success in this phase satisfyingly concludes our last phase in the protocol development cycle of our *Natural Engineering* approach. The chapter is based on the following technical report.

- Horst F. Wedde and Muddassar Farooq. *Bee Inspired Linux Routing Framework: A Step from Swarm Intelligence to Natural Engineering*. Technical Report 803, Department of Computer Science, University of Dortmund.

**Chapter 6–Conclusion and Future Work.**
In this chapter, we summarize the contributions of our work. We stress the need for the *Natural Engineering* approach because this significantly helped us in successfully designing a dynamic, simple, efficient, robust, flexible and scalable multi-path routing algorithm and then installing it in a real network of Linux routers. We believe that a similar approach can help in realizing other Nature inspired algorithms in their respective real environments.
We conclude the chapter with interesting future directions. The most important one is: design and development of a dedicated Nature inspired router in hardware which optimally runs Nature inspired routing algorithms. Before this step is taken, we have to reengineer *BeeHive* in such a fashion that it is capable of seamlessly replacing *OSPF* in the existing packet switched IP networks.

# 2

# A Comprehensive Survey of Nature Inspired Routing Protocols

*The major contribution of the chapter is a comprehensive survey of existing state-of-the-art Nature inspired routing protocols developed by researchers who are trained in novel and different design doctrines and practices. Nature inspired routing protocols have been becoming the focus of research because they achieve the complex task of routing through simple agents which traverse the network and collect the routing information in an asynchronous fashion. Each node in the network has a limited information about the state of the network, and it routes data packets to their destination based on this local information. The agent-based routing algorithms provide adaptive and efficient utilization of network resources in response to changes in the network catering for load-balancing and fault management. The chapter describes the important features of stigmergic routing algorithms, evolutionary routing algorithms and artificial intelligence routing algorithms for fixed telecommunication networks. We also provide a summary of the protocols developed by the networking community. We believe that the survey will be instrumental in bridging the gap among different communities involved in research of telecommunication networks.*

## 2.1 Introduction

The design and development of multi-path, adaptive and dynamic routing algorithms has been approached by different communities of researchers, each having a strict traditional design philosophy, leaving little room for cross-fertilization of novel ideas between different research communities. This provided us the grist for the mill for providing a comprehensive survey of routing protocols, designed and developed by different communities of researchers, for different types of telecommunication networks: circuit-switched and packet-switched. The major objectives of the survey are:

- to understand the basic design concepts and doctrines of the different communities, and then contemplating the strengths and short-comings of each approach.
- to create awareness among the researchers about state-of-the-art routing algorithms developed by other communities.
- to create a vision about future directions/challenges for routing protocols as they may be employed in totally different operating environments like sensor networks.
- to allow for cross-fertilization of ideas which will help in taking a comprehensive approach to counter the challenges of complex large-scale telecommunication networks.
- to create an intelligent and knowledge-aware network layer implicitly taking care of network management and traffic engineering, by virtue of its intelligent routing algorithms.

- to lay the ground for a comprehensive performance evaluation framework, for the purpose of comparative evaluation of routing protocols.

### 2.1.1 Organization of chapter

The rest of the chapter is organized as follows. Section 2.2 will provide major challenging requirements that a routing protocol should be able to meet, then giving rise to a taxonomy of routing protocols in Section 2.2.2. We will first provide an overview of Ant Colony Optimization (ACO) metaheuristic in Section 2.3, and then discuss in detail different routing algorithms inspired from ACO. Section 2.4 will outline important features of Evolutionary Algorithms (EA) and then describe corresponding routing algorithms. Subsequently, we will conclude our survey of routing protocols for fixed networks in Section 2.5. We will briefly discuss the state-of-the-art routing protocols that are based on the traditional design paradigm of distance vector or link-state routing methods. Finally, we conclude our survey by emphasizing the cross-fertilization of design principles of different approaches, for the purpose of a comprehensive approach to solutions for the challenges of modern telecommunication networks.

## 2.2 Network routing algorithms

In this section, we briefly outline the challenges facing the telecommunication sector because of an ever increasing demand for intelligent/integrated multimedia services from the user community. The solutions to such challenges lie in a multi-dimensional landscape of requirements for designing, developing and implementing intelligent routing algorithms. These features are summarized in Section 2.2.1. In Section 2.2.2, we will outline a taxonomy of routing algorithms according to several criteria, reflecting different design doctrines, switching strategies, and network environments.

### 2.2.1 Features landscape of a modern routing algorithm

The design goals of a routing algorithm are summarized in the following:

- *Optimality* of a routing algorithm could be defined as the ability to select the best route [38]. The best route could be defined in terms of a quality metric, which in turn might depend on a number of parameters i.e. hops, delay or a combination of both. A routing algorithm can easily compute a best path in a static network but it becomes a daunting task in a dynamic network.

- *Simplicity* is a desirable feature of any routing algorithm. A routing algorithm should be able to accomplish its task with a minimum of software and resource utilization overhead. Simplicity plays an important role when a routing algorithm has to run on a computer with limited physical resources [38].

- *Robustness* of a routing algorithm could be described as its ability to perform correctly in the face of unusual or unforeseen situations like hardware failures, high load conditions and incorrect implementations [38]. A router has to quickly react to the anomalies and re-route the packets on alternative paths. This property is also known as *fault-tolerance*.

- *Convergence* is the process of agreement, by all routers, on optimal paths. In face of router failures, a routing algorithm should be able to make all routers quickly agree, through transmitting update messages, on alternative optimal routes. Routing algorithms that converge slowly can cause loops or network outages [38].

- *Flexibility* is the ability of a routing algorithm to quickly and accurately adapt to a variety of network circumstances. They should be programmed to adapt to changes in the available network bandwidth, routers' queue size, and network delay, among other variables [38].

- *Scalability* is the ability of an algorithm to operate in large networks without an associated increase in demand for software/physical resources and resource utilization overhead. The control packets should occupy a small bandwidth, they should have small processing overhead and routing tables should occupy small memory etc.

- *Multi-path Routing* exploits the resources of the underlying physical network, by providing multiple paths between source/destination pairs [118]. This requirement allows the protocols to achieve higher transfer rates than given by the bandwidth of a single link. Multi-path feature also helps in doing load balancing in the face of congestion, allowing for delivering more packets with smaller delays at the destination.

- *Reachability* is the ability of a routing algorithm to find at least one path between each source/destination pair [198].

- *Quality of Service (QoS)* is the ability of an algorithm to administer better service to selected real time traffic like multimedia by providing dedicated bandwidth, controlled jitter and latency [38].

### 2.2.2 Taxonomy of routing algorithms

Routing algorithms have been classified in [73] according to criteria reflecting upon fundamental design and implementation options like

- *Structure.* Are all nodes treated equally in the network?

- *State Information.* Is network-scale topology information available at each node?

- *Scheduling.* Is routing information continually maintained at each node?

- *Learning model.* Do packets or nodes have an intelligent learning model?

- *Queue control.* Do nodes employ load-balancing to manage growth of queues?

Such issues could be raised and discussed under all following dimensions of networking as they are grouped below, under the topics routing strategy/policy, design doctrine, specific aspects of telecommunication networks:

#### Routing strategy/policy

Here we provide only a brief overview explaining the concepts of the taxonomy in [38].

- *Static versus Dynamic.* Static routing algorithms are simple table mappings established by network administrators before the routing begins. Such algorithms can react to changes only if the network administrator alters these mappings based on his experience with traffic patterns in the network.
  Dynamic algorithms update their routing tables according to changing network circumstances by analyzing incoming routing update messages and rerunning the algorithms to calculate new routes. This feature makes them suitable for todays large, constantly changing networks.

- *Single-Path versus Multi-path.* Single-Path routing algorithms determine the best path to a destination while multi-path routing algorithms discover and maintain multiple paths to a given destination. This feature allows them to multiplex the traffic to the destination on multiple paths, as a result, both their throughput and reliability are higher than in case of single-path routing algorithms.

- *Flat versus Hierarchical.* Flat routing algorithms consider all nodes in the network to be peers and they maintain an entry in their routing tables for all routers. This allows peers to discover a best route at the cost of transmitting more control packets and maintaining larger routing tables. Hierarchical routing algorithms form a logical group of routers and organize them into areas, domains and autonomous systems. Such algorithms require two types of routers, *intra-domain routers*, which route traffic within a domain, and *backbone routers*, which route traffic between domains. The advantage of such organization is that it mimics the traffic patterns of organizations in which most of communication occurs within small areas like factory locations in a big company. So each location could work with simple intra-domain routing algorithms. In this manner such organization requires significantly smaller routing tables which, in turn, require smaller memory storage and little waste of bandwidth for maintaining routes.

- *Intradomain versus Interdomain.* Intra-domain routing algorithms route data packets within the same domain only while inter-domain routing algorithms route data packets between domains. Within a domain or *Autonomous System (AS)*, system administrators could select their own routing policy. Due to the different nature of such algorithms, an optimal intra-domain routing algorithm may not necessarily be an optimal inter-domain routing algorithm.

- *Link-State versus Distance Vector.* In *links-state algorithms* each node floods the status of its links to all nodes of the network. Then each router constructs a graph of the complete topology and applies the Shortest Path First routing algorithm for obtaining the next hop on a shortest path to each destination and storing it in its routing table. In *distance vector* algorithms, routers send updates only to their neighbors. *Link-state algorithms* converge quickly, scale better but require more CPU power and memory than distance vector algorithms, therefore, they are expensive to implement and support.

- *Host Intelligent vs Router Intelligent.* In *host intelligent algorithms* a host determines the entire route to a destination and appends it as a header to each packet, known as *source routing*. Other routers in the system simply forward the packets to the next hop contained in the header of the packet. In *next hop routing algorithms* routers are intelligent and they discover and maintain paths while executing their algorithms, therefore, they are termed as *router intelligent algorithms*.

- *Global vs Local.* In *global routing algorithms*, each node requires the information about all nodes, their inter-connectivity and cost of links for constructing a graph and then applying path finding algorithms on it. In contrast, *local algorithms* do not have access to information about the complete topology, rather they work with a local traffic model, maintained at each router, for reaching at a routing decision.

- *Deterministic vs Probabilistic. Deterministic algorithms* associate, for every destination in the routing table, an outgoing interface identifier and a cost associated with choosing that interface. *Probabilistic algorithms* associate probability values to all neighbors of a node, through which a packet could reach its destination, depending on the costs of the links to the neighbors. A neighbor with a higher probability value is supposed to be on a better path than a neighbor with a lower probability value. The probabilities of all neighbors are normalized such that their sum always remains one. *Probabilistic algorithms* multiplex the network traffic on different paths, depending on their probability value, and hence have better performance than *deterministic algorithms*, but they require more memory and CPU power [198].

- *Constructive vs Destructive. Constructive algorithms* begin with an empty set of routes and incrementally add routes till final routing tables have been constructed. In contrast *destructive algorithms* start with a fully connected graph as an initial condition in which all routes are available, and gradually those paths are removed from the routing tables which do not exist in the network [202].

- *Best effort vs QoS. Best effort algorithms* do not provide any guarantee that the demands of the applications would be met while *QoS algorithms* reserve the resources in the network to meet the demands of the applications. *QoS algorithms* provide guarantees to the applications through a policy of admission control.

**Design doctrine**

Routing algorithms could alternatively be classified on the basis of the design philosophy of their developers. The researchers in each community have been trained with a certain design and analysis doctrine which leaves little room for cross-fertilization of ideas from other communities. In this subsection, we briefly provide an overview of these communities that will help the reader in understanding the design principles of different types of routing algorithms. Given a mutual understanding of the various backgrounds of these communities there is a chance for developing state-of-the-art routing algorithms for the networks of the new millennium. We have categorized important routing algorithms according to their design doctrine in Figure 2.1. This figure can also be used as a road map for our survey of routing protocols for fixed telecommunication networks. The communities are discussed in the sequel:

(a) The *Networking community* has pioneered the work in the field of packet switched networks. The roots of this work go back to the development of ARPANET and a novel routing algorithm, which is based on an asynchronous Bellman-Ford algorithm [15, 136]. Later on many dynamic and multi-path routing algorithms have been developed by following the classic methodology for routing protocol development: non-intelligent link-state packets are used to collect information about the costs of neighbors and then to propagate them in the whole network. Consequently, they all suffer from the same shortcomings: "wrong" or "out-of-order" local estimates have a global impact [48], and the algorithms require a global system model to execute Dijkstra's shortest path algorithm [44]. The algorithms could be classified as global and deterministic routing algorithms.

(b) The *Artificial Intelligence Routing community* works in two different areas: *Machine Learning* and *Agent-based Learning*. The first community uses Reinforcement Learning (RL) [104] techniques, developed as a branch of *Machine Learning*, in order to propose routing algorithms for packet switched networks. Examples are Q-routing [23] and PQ-routing [37], both are based on Q-learning [212, 213]. Such algorithms are adaptive, decentralized, dynamic, local and deterministic. Agent-based learning methods resulted in specific routing algorithms [199, 44, 89, 132, 48]. The major advantages of such algorithms are summarized as follows:

    - The algorithms do not require an a priori global system model of the network, rather they utilize a local system model as observed by the agents.
    - The agents gather the network state in a decentralized fashion and leave the corresponding information on visited nodes. This enables them to make routing decisions in a decentralized fashion, without the need of a global controller.
    - The algorithms have the ability to adapt autonomously to changes in the network, or in traffic patterns.
    - The management of the network comes as a complimentary benefit of using such mobile agents.

    The major emphasis of such routing algorithms is on designing intelligent agents for doing routing, management and control of networks *in an autonomous manner*. The multi-agent systems provide a good infra-structure for design and development of such mobile agents [188, 227, 85, 199, 89, 114, 132], however, the intelligence is achieved at the cost of complex design paradigms [95, 32, 239, 238, 149, 246, 247, 98, 100, 25, 59, 177].

(c) The *Natural Computing* research has two major directions: Evolutionary computing [82] [96] and Swarm Intelligence [17]. Evolutionary computing takes the evolution process in living cells as a basis for developing algorithms/systems. Consequently, evolutionary routing algorithms employ the evolutionary operators of *selection, cross-over* and *mutation* for on-line adaption to cope with changes in network environments. *DGA* (Distributed Genetic Algorithm) [120] is one such routing algorithm. The second emerging area, Swarm Intelligence, studies different self-organizing processes in Nature and utilizes their principles as an inspirational metaphor to propose novel solutions to different daunting classical scientific problems. The novelty comes again from the fact that such systems lack one central complex controller, which normally co-ordinates/schedules different tasks in the system, by virtue of its access to the global system state. On the contrary, these population-based systems have simple entities that have only local knowledge but *together* they form an intelligent system [17], [196]. *ABC* [168], *AntNet* [52], and *BeeHive* [221] belong to this class of routing algorithms. Nature inspired routing algorithms are mostly adaptive, decentralized, local, dynamic and probabilistic.

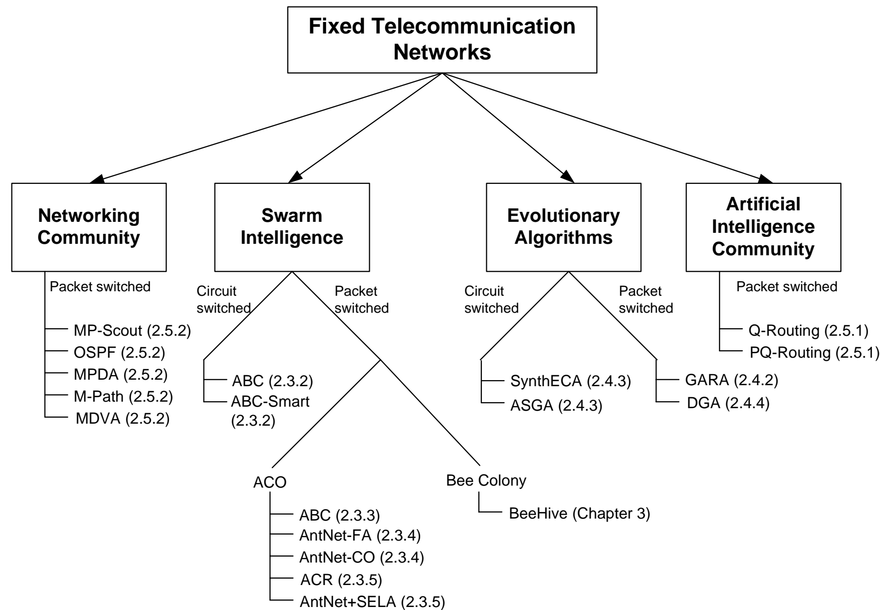



Figure 2.1: A taxonomy of routing protocols for fixed telecommunication networks

### Specific aspects of telecommunication networks

We will restrict our survey of routing algorithms to only two types of telecommunication networks namely, connection-oriented circuit switched networks and connection-less packet switched networks. We also focus on the *Natural Computing* algorithms for these two types of the networks, however where appropriate, we will provide a brief summary of the algorithms developed by other communities.

Each type of network comes with a different set of requirements that a routing algorithm should meet. Topology changes are less frequent in fixed telecommunication networks but the traffic patterns are non-deterministic. Therefore a routing algorithm should be able to do congestion control. In connection-oriented networks, a circuit is reserved for each connection between a pair of source and destination, therefore, a routing algorithm should have good admission control through efficient resource utilization, in order to reduce the call blocking probability.

## 2.3 Ant Colony Optimization (ACO) routing algorithms for fixed networks

In this section, we first briefly summarize important elements of the ACO metaheuristic in Section 2.3.1, and then provide a survey of two state-of-the-art routing algorithms designed on the basis of ACO metaheuristic: *ABC* (Section 2.3.2), which is designed for circuit switched telecommunication networks, and *AntNet* (Section 2.3.4), which is designed for packet switched telecommunication networks.

### 2.3.1 Important elements of ACO in routing

The Ant Colony Optimization (ACO) metaheuristic has been inspired by operating principles of ants [18], which empower a colony of ants to perform complex tasks like nest building and foraging [64]. We summarize important elements of ACO, which have been utilized in routing algorithms, in the sequel.

#### Stigmergy

The ants are able to find the shortest path from their nest to a food source by sharing information through *stigmergy* [64, 47]. Stigmergy is a form of communication in which social insects like ants communicate indirectly through the environment [84, 47]. Ants lay pheromone while foraging. As a result, the concentration of pheromone on the shortest path is reinforced at a higher rate than the other paths. Ants tend to prefer higher pheromone concentration paths, which results in a majority of ants using a shortest path for foraging in a steady state [47]. Stigmergy is the most important element of the ACO metaheuristic and has been instrumental in developing a society of mobile ant agents, as they cooperate in solving discrete optimization and control problems [64, 63, 62, 67, 124, 65]. Here we limit our survey to applications of ACO to telecommunication networks.

#### Pheromone control

Bonabeau et al. have pointed out in [18] that the success of ants in collectively locating shortest paths is only statistical. If many ants initially happen to choose a non-optimal shortest path, other ants will follow this path which will result in pheromone reinforcement along this path. Consequently, ants will travel on a stagnating non-optimal path in a steady state. However, if we assume that ants do find shortest path in a steady state even then this stagnation is not helpful because if all packets follow the shortest path then this will lead to congestion on this path. Consequently, the path becomes non-optimal and other non-optimal paths may become optimal due to changes in network conditions, or due to discovering of new paths after changes in the topology [174]. Therefore, it is extremely important to counter stagnation through intelligent pheromone control strategies. We outline some of these strategies here, however, the interested reader will find a detailed discussion in [174].

- *Evaporation.* In ACO algorithms, the values of pheromone $t_{ij}$ in all links is decreased by a factor $p$ such that: $t_{ij} \leftarrow t_{ij}(1-p)$ [63]. This helps in reducing the influence of past experience during decision making.

- *Aging.* The amount of pheromone that an ant lays on a path decreases with its age, an older ant lays less pheromone than a younger one [169]. Since ants mostly assume symmetric links (in which cost of links in both directions are the same). The solution for asymmetric links is that ants measure the costs during a forward trip and deposit pheromone on a backward trip.
  *Evaporation* and *Aging* favor present experiences which result in discovery of new paths by avoiding stagnation.

- *Limiting and smoothing pheromone.* Some authors circumvent the problem of stagnation by setting an upper bound $t_{max}$ of pheromone for every edge (i,j) [189]. This reduces the preference of ants for optimal paths over non-optimal paths. However, one should be careful that *pheromone limiting*, if not used in conjunction with *evaporation*, will make all paths equal once the pheromone value reaches $t_{max}$ for all links. *Pheromone smoothing* ensures that only a small amount of pheromone is permitted on paths where the current pheromone concentration is closer to $t_{max}$. Consequently, a few dominant paths are not generated. But this feature might lead to a situation in which the number of ants that prefer to select a non-optimal path keep increasing because ants deposit more pheromone on these non-optimal links, even though an optimal path might remain optimal in a steady/stable state.

- *Pheromone-heuristic control.* The authors of [65, 67] use the amount of pheromone in a link combined with a heuristic function to influence the decision of an ant. The heuristic function $n_{jd}$ for telecommunication networks is determined by the queue length $q_j$ (number of bits on network interface of neighbor $j$ in a router). Finally, this heuristic, based on the current state of the network, is combined with a long-term *learned goodness* $p_{jd}$ of using neighbor $j$ for reaching a destination $d$ [52]. Such a hybrid approach enables a routing algorithm to be responsive to transient changes in the networks. However, setting the weight value in the formula in which the heuristic factor is combined with the long term learned goodness of a link is very sensitive. If a policy exceedingly emphasizes the weight of heuristic component then it might cause oscillations, and inadequate emphasis would make the algorithm not to react to transient changes in the networks.

- *Privileged pheromone laying.* The authors of *AntNet* [52, 55] enhance the ACO metaheuristic by a concept of *privilege pheromone laying.* In their algorithm, ants first evaluate the quality of their solution and then deposit the amount of pheromone based on the quality. They model the quality of a solution as a function of the trip time of a forward ant, the best known trip time, and few other statistical parameters. The experiments reported in [52, 55] reveal that such a policy results in better convergence and performance. Later on the authors of [189] devised the *FDC fitness landscape* approach, which compares the fitness of a solution of each ant with an optimal solution and then deposits the amount of pheromone. The experiments reported in [189] confirm that *FDC* contributes to obtaining accurate and better results.

### 2.3.2 Ant-based control (ABC) for circuit switched networks

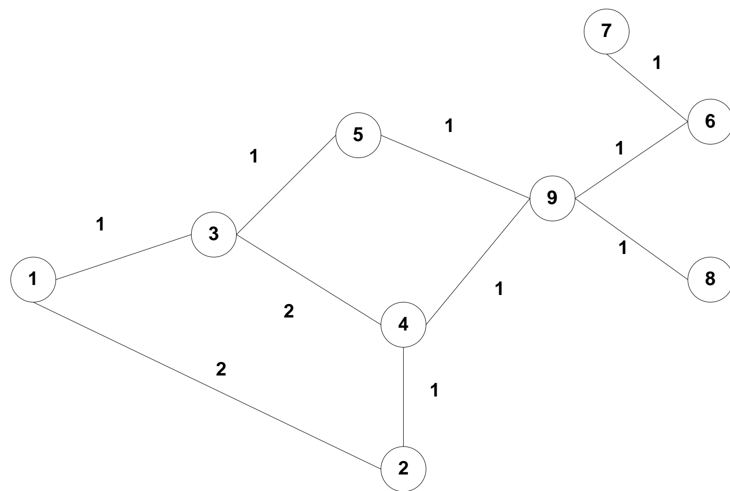
Schoonderwoerd et al. [169, 168, 167] were the first to apply the ACO metaheuristic to routing and load-balancing problems in circuit-switched telecommunication networks. As a symmetric network, a circuit-switched network reserves a virtual circuit between a sender and a receiver by explicitly connecting them through cross-bar switches. Consequently, the major challenge is to distribute the calls over multiple switches so that the system can support a maximum number of possible calls during peak hours. Such a network is not able to admit a call if all input ports of a cross bar switch are connected to its output ports. Consequently, congestion could be defined as a function of the number of used connections in a cross bar switch [76, 77, 162]. The performance of a switching algorithm is measured in terms of the number of calls which are blocked or failed due to congestion [6].

In the *ABC* algorithm, each node in the network stores the following attributes [169]:

- The capacity is the number of simultaneous calls that a node (cross-bar switch) can manage. The remaining free capacity of a switch is also stored.

- A pheromone based routing table in which probability values, representing goodness of a node's neighbors for reaching each destination are stored. Each row $i$ in the table represents a destination and each column $j$ represents a neighbor. Each probability value $p_{jd}$ represents the goodness of choosing $j$ as a next hop for reaching destination $d$.

- A probability value of this node being the end point of a call.




| | 4 | 5 | 6 | 8 |
|---|---|---|---|---|
| 1 | $p_{41}$ | $p_{51}$ | $p_{61}$ | $p_{81}$ |
| 2 | $p_{42}$ | $p_{52}$ | $p_{62}$ | $p_{82}$ |
| 3 | $p_{43}$ | $p_{53}$ | $p_{63}$ | $p_{83}$ |
| 4 | $p_{44}$ | $p_{54}$ | $p_{64}$ | $p_{84}$ |
| 5 | $p_{45}$ | $p_{55}$ | $p_{65}$ | $p_{85}$ |
| 6 | $p_{46}$ | $p_{56}$ | $p_{66}$ | $p_{86}$ |
| 7 | $p_{47}$ | $p_{57}$ | $p_{67}$ | $p_{87}$ |
| 8 | $p_{48}$ | $p_{58}$ | $p_{68}$ | $p_{88}$ |

**Routing Table at Node 9**

Figure 2.2: Pheromone routing table in ABC

In *ABC*, an ant, launched by node $s$ and traveling towards destination $d$, will update the probability values for its source node at each intermediate node passed. We now refer to Figure 2.2 to illustrate the relevant aspects of the *ABC* algorithm. Let us assume that an ant has been launched by node 3, and its destination is node 9. Once the ant reaches node 5, it will update the $p_{33}$ value in the routing table and once it reaches at node 9 then it will update the $p_{53}$ in the routing table of node 9. This will influence the ants which are traveling towards node 3 and are passing through node 9. This approach, therefore, has been specifically designed for symmetric links [52, 174].
Schoonderwoered et al. used *aging*, *delaying* and *noise* techniques to counter stagnation in the probability values in the routing tables. The amount of pheromone $\delta p$ that an ant is allowed to deposit is given by the formula $\delta p = \frac{0.08}{age} + 0.005$. The purpose of *delaying* is to increase the transit time of certain ants proportional to the spare capacity of the node. The *delay* is defined as $delay = 80 \times e^{-0.075 \times r}$ where r is the remaining capacity of a node. Consequently, the rate at which ants are transmitted from congested nodes is reduced, and due to aging mechanisms the ants deposit less pheromone on the nodes, which they subsequently visit. In this way the influence of the ants, which visited a congested node, on other ants is reduced. Finally, a certain ratio of ants do not follow the paths according to the pheromone values in the routing tables. Rather they follow random paths where they may discover new and better routes in dynamic networks. Schoonderwoered et al. have experimentally verified that the *ABC* algorithm, on the average, drops less calls as compared with the algorithm of Appleby and Steward [6]. Moreover, it quickly reacts to changes in the topology.
Bonabeau et al. extended *ABC* with the idea of *smart agents* [19], which utilize the concept of dynamic programming: the agents update the probability values for all visited nodes, at a given node, rather than just for their source node. Consider e.g. an ant agent launched from node 1 (see Figure 2.2) and traveling towards node 9 via nodes 3 and 5. At node 3 it will update the probability value $p_{11}$, at node 5, it will update the probability values $p_{31}$ and $p_{33}$ and at node 9 it will update the probability values $p_{51}$, $p_{53}$ and $p_{55}$. Consequently, *ABC* with *smart agents* reduced the number of calls which were dropped as compared with *ABC* and it was also able to react to changes in the topology. However, *smart agents* use a similar policy as used by ants in *ABC* for updating the routing tables. Compared to *ABC* agents, *smart agents* have a more complex behavior but the objective is achieved with fewer agents.
The authors of [164] have studied the behavior of *ABC* on a different network topology and confirmed the earlier results published by the authors of *ABC*. Recently they enhanced the original *ABC* in their work reported in [163]: if the age of an ant that arrived at a node is greater than the current maximum age stored at the node then it decreases the goodness (pheromone) value rather than increasing it. This concept is known as *anti-pheromone*. They also employed the probabilistic routing method as used for phone calls on a topology of 25 nodes. Their modified algorithm has shown a certain degree of improvement as compared with original *ABC*.

### 2.3.3 Ant-based control (ABC) for packet switched networks

Subramanian et al. [191] developed an algorithm for packet-switched networks on the basis of the ideas of *ABC*. They designed two types of ants: *regular* and *uniform*. Regular ants update the pheromone values in the routing tables based on the accumulated cost of traveling to a node. In Figure 2.2, an ant traveling from node 3 to node 9 via node 5 will update the value $p_{53}$ at node 9 based on the accumulated cost of the path from node 3 to node 5 and then from node 5 to node 9. Uniform ants randomly select their next hop and they update the pheromone values in the routing tables based on the costs in the direction opposite to their travel. A uniform ant traveling from node 3 to node 9 will update the value $p_{53}$ at node 9 based on the cost of link from node 9 to 5 and node 5 to 3. The algorithm assumes that each node has determined the cost information of the link to its neighbors.
Heusse et al. [93], based on ideas of *ABC*, proposed an algorithm with *cooperative asymmetric forwarding (CAF)* for routing in packet switched networks. Their basic idea is: once a data packet is traveling from node 5 to node 9 then it carries with it the cost of link $c_{59}$, which is a sum of waiting and propagation delay, from node 5 to node 9. At node 9, it leaves this value in a reverse

routing table. Once an ant, which is traveling from node 6 to node 3 via node 9 arrives at node 9 and selects neighbor 5 as a next hop then it carries with it this value. At node 5, it adds this cost to $c_{96}$ to determine the accumulated cost $c_{56}$. It carries with it the estimates of reaching all nodes which it had visited, and then updates all corresponding entries. At node 5, it will update $p_{99}$ and $p_{96}$ depending upon $c_{59}$ and $c_{56}$. The algorithm will not work properly under low traffic load scenarios in which a small number of data packets are sent on the network. As a result, an ant will carry old values in the reverse routing tables which might degrade the performance of the algorithm. Moreover, an additional reverse table is required to be maintained.
Van der Put and Rothkrantz [201, 200] designed *ABC-backward* based on the concept of forward and backward moving ant agents. The algorithm applies *AntNet* concepts (to be introduced shortly) to *ABC*. The algorithm can be used on cost asymmetric networks. The authors have experimentally verified that *ABC-backward* has a better performance than *ABC* on both cost symmetric and cost asymmetric networks. *ABC-backward* solved a serious fax-distribution problem faced by KPN telecom (largest telephone company in Netherlands).

### 2.3.4 AntNet

*AntNet* was proposed by Di Caro and Dorigo in [51], [54], [50], [52]. It is inspired by the principles of the ACO metaheuristic but has additional network specific enhancements as well. The algorithm is designed for asymmetric packet-switched networks, and the primary objective of the algorithm is to maximize the performance of a complete network. The algorithm implicitly achieves load balancing by probabilistically distributing packets on multiple paths.
In *AntNet* the network state is monitored through two ant agents: *Forward_Ant agent and Backward_Ant agent*. The agents are equipped with a stack on which node address and the trip time estimate to the nodes are pushed. A Forward_Ant agent is launched at regular intervals from a source to a certain destination depending upon the amount of traffic generated for the destination at the source. The probability $p_{id}$ for launching a Forward_Ant agent to destination $d$ at node $i$ is $p_{id} = \frac{f_{id}}{\sum_{k=1}^{D} f_{ik}}$, where $f_{id}$ is the number of bits flowing from node $i$ towards node $d$ and $D$ is total number of nodes in the network. Forward_Ant agent uses the normal queues to experience the true network conditions. If a Forward_Ant agent follows a cyclic path then the data about the nodes which lie on the cyclic path are removed from the stack. However, the agent is allowed to explore the network if the time it spent in the cycle is less than half the Forward_Ant agent lifetime. Once Forward_Ant agent reaches its destination, it creates a Backward_Ant agent and transfers all information to it. Backward_Ant agent visits the same nodes as Forward_Ant agent yet in a reverse order, and it modifies the entries (deposit of pheromone) in the routing tables in accordance with the trip time from the nodes to the destination. A Backward_Ant agent is only allowed to update entries in the routing tables of the intermediate nodes if it discovers a good sub-path from the intermediate node to the destination. The goodness is defined based upon the trip time. The trip time values are calculated by taking the difference of entrance times of two subsequent nodes pushed onto the stack. The updating of routing tables only influences data packets and Forward_Ant agents, which are traveling from node $i$ to node $d$. The nodes in *AntNet* maintain the average trip times, the best trip times, and the variance of the trip times for each destination. In this way, information is statistically maintained at each node in the network, for subsequent routing decisions. Backward_Ant agent uses the system priority queues so that it quickly disseminates the information to the nodes.
*AntNet* uses the heuristic function $l_j = 1 - \frac{q_j}{\sum_{k=1}^{N} q_k}$, where $q_j$ is the number of bits in queue of neighbor $j$ and N is the total number of neighbors. The heuristic function favors neighbors with smaller queue lengths. $P_{jd}$ is the goodness of neighbor $j$ for reaching destination $d$. Backward_Ant agent enhances the goodness of neighbor, from where it arrived, using the formula $P_{jd} \leftarrow P_{jd} + r(1 - P_{jd})$, where r is a reinforcement factor, and it decreases the goodness value of other neighbors using the formula $P_{kd} \leftarrow P_{kd}(1-r)(k \neq j)$. $P_{jd}$ is a long-term learned value which provides an insight about the goodness of a neighbor for a particular destination. The reinforce-

ment factor is defined as a function of the current trip time, the best trip time and the statistical confidence intervals. Finally, this $P_{jd}$ value is combined with the heuristic value $l_j$ (just defined) to react to current state of the network using the formula $P'_{jd} = \frac{P_{jd}+\alpha l_j}{1+\alpha(N-1)}$, where $\alpha$ weighs the heuristic function with the probability values stored in the routing tables.
*AntNet* applies the concept of stochastic spreading of data packets along all paths according to the goodness of the paths. However, the goodness, $P_{jd}$ is further rescaled to reward better goodness solutions more than ones of lower quality. The rescaled values are stored in an another table, which is used during the switching of data packets. The concept of using two tables, one for ant agents and another for data packets has been elaborated in [48].
Di Caro and Dorigo have conducted a number of experiments on different topologies like simpleNet, NSFNet and NTTNet which are reported in [51], [54], [50], [52]. They have chosen *throughput* and *90th percentile of packet delays as the performance* parameters. The experiments reported have shown that *AntNet* outperforms, with respect to throughput and delay, all other competitors, which consist of Q-routing, PQ-routing, Shortest Path First (SPF) and *OSPF*, except the Daemon algorithm. The improvement in performance is achieved at a cost of less than 1% of the bandwidth occupied by ant agents.
The authors proposed a variant of *AntNet*, known as *AntNet-FA* or *AntNet-CO*, in [55]. In *AntNet-FA*, Forward_Ant agents do not have to wait in the queue to measure the queuing delay. Rather they use an estimation model to estimate the delay. This feature allows a Forward_Ant agent to use priority queues as well. The estimation model estimates the trip time $t_{ij}$ from node $i$ to $j$ using the formula $t_{ij} = pd_{ij} + \frac{q_j}{b_{ij}}$, where $pd_{ij}$ is the propagation delay of the link from node $i$ to $j$, $q_j$ is queue length in bits of neighbor $j$ at node $i$ and $b_{ij}$ is the bandwidth of the link from node $i$ to $j$. Such a policy facilitates the quick spreading of the routing information specially in large topologies. The authors have reported in [55] that the performance of *AntNet-FA* is significantly better than *AntNet* on a 150 node topology.
The *AntNet* algorithm utilizes the concept of privileged pheromone laying along with heuristic pheromone control to react to changes in the traffic patterns. Let us assume that a Forward_Ant wants to find a path from node 9 (please see Figure 2.2) to node 2. Interestingly, *AntNet* will maintain the routing table entries $p_{62}$ and $p_{82}$ for reaching node 2, although it is impossible to reach node 2 via node 6 and node 8. Consequently, $p_{62}$ and $p_{82}$ will be zero. However, it might be possible that $q_6$ and $q_8$ are significantly smaller than $q_5$ and $q_4$. Therefore, it is possible once $p_{62}$ and $p_{82}$ are combined with a heuristic value then new values for $p_{62}$ and $p_{82}$ will be non zero. This will lead to sending data packets with destination 2 to node 6 and node 8, which of course is an error in the algorithm, but to our knowledge the problem has never been addressed to date. After subsequent loops, a Forward_Ant or a data packet may ultimately reach node 2 through nodes 5 or 4.
The ant agents, by utilizing stack and making forward and backward trips, may occupy a significantly large portion of bandwidth, in comparison to ant agents of *ABC* in large topologies. The agents perform complex computations once they arrive at a node. As a result, the processing complexity of ant agents will be significantly higher than the ant agents of *ABC*. However, we believe that a thorough experimental study needs to be done to evaluate the scalability of *AntNet* in large topologies.

### 2.3.5 Ant Colony Routing (ACR) and AntNet+SELA QoS aware routing

Di Caro has discussed ACR in [48], which is a general framework for designing fully distributed and adaptive systems for network control and management. ACR can be viewed as a distributed society of static agents, which are known as node managers, and mobile agents, which are proactively or reactively launched in the network. Node managers autonomously manage node activities by learning and then following stochastic management policies based on local pheromone values, which represent goodness of different control actions. However, they expand their "sensory field" to acquire information about their environment with an adaptive generation of mobile agents. Mobile

agents take an active *preceptors* role on behalf of the node managers which launch them. These agents collect the important parameters which act as input parameters to learning strategies of node managers. A node manager, based on the feedback provided by preceptor agents, might alter its control actions. Stochastic decision policies are well suited for non-stationary and distributed environments because they help in spreading data over multiple paths thus implicitly providing load-balancing as well. A preceptor agent may either follow a point-to-point mode by following pheromone tables for already discovered destinations, or it might follow a broadcast strategy for a destination which is not known at a node. The broadcast strategy helps in replicating the active perceptions to discover as many good options as possible. Active preceptors are situated at a lower level in the hierarchy than node managers, and hence must not be allowed to directly modify internal state of the node managers. Preceptors simply communicate to the node managers the collected information, and they may accept or reject the information. This contributes to designing secure systems, in which malicious agents could not directly modify the internal state of node managers, and to practicing an object oriented design (information hiding). *Effectors* are mobile agents which have a deterministic precompiled behavior. They are used to carry out highly specialized tasks like allocation/deallocation of resources. In QoS routing, this will help in finding the paths to a destination in which enough network resources can be reserved to meet the QoS guarantees for the application. As a multi-path routing system ACR will obviate to discover new paths in case of router failures since "backup" paths are already available. In this way, multiple ant colonies can coexist together and manage the activities of a node in a social agreement with all other nodes.
AntNet+SELA [56] was designed to provide QoS guarantees to a variable bit rate (VBR) traffic in ATM networks. ATM networks provide statistical guarantees by reserving virtual circuits either on a per flow basis or on a per destination basis. The node managers which do both admission control and routing are designed as *stochastic estimator learning automata* (SELA) [150] to be applied to a distributed routing system for ATM networks [7]. The node managers utilize active preceptors to proactively update a link-state database in which the goodness values of different paths leading to a destination are stored. Active preceptors utilize different routing tables than data packets. When a new application arrives at a node then two groups of active preceptors are launched. The first group consists of path-probing setup ants which probe k different paths leading toward the destination, and the second group consists of path-discovering setup ants which make use of existing pheromone tables to discover new QoS-feasible paths for the applications. Both agent groups reserve resources temporarily as they traverse the paths. Path-discovering setup ants bias their decision on the current status of queue lengths to calculate the goodness of a neighbor (the value of $\alpha$ is set much higher (please refer to Section 2.3.4)). Moreover, they always choose the link which has the highest probability value. If the probability values of two best links differ minimally then the ants are allowed to replicate themselves on both links, however, replication is allowed only once for a better control of the number of ants. If a traversed path does not meet the QoS requests, an effector ant is generated which follows the same path but in reverse order to free the allocated resources. However, if an ant is able to locate a QoS feasible path to a destination then it comes back to the source node and provides information about the path to the node manager. If multiple ants have come back then a node manager may decide whether to split the traffic load from the application on multiple paths or not. Now the application can start sending the packets. Effector agents (monitor ants) are periodically launched, once the application is running, to provide feedback about the state of the links to the node managers. The feedback is helpful in doing load-balancing if the current network load is not balanced. Once the application is finished then effector agents are launched over the paths used by the application to free allocated resources. The management scheme can handle QoS and best-effort traffic at the same time (by utilizing routing tables built by proactive monitor ants).

### 2.3.6 A brief history of research in AntNet

Oida and Kataoka [145] decided to improve an earlier version of *AntNet* in which the status of data link queues was not used in the goodness formula (yielding no pheromone heuristic control).

Without this queue dependency feature, *AntNet* will suffer from stagnation once the goodness of any link of a neighbor reaches 1. The authors of [145] modified the routing table updating rules to avoid the "locking" of routing tables. Their algorithms *DCY-AntNet* and *NFB-Ants*, upon comparison with an earlier version of *AntNet* [49], performed much better under challenging situations. Doi and Yamamura [60, 61] also proposed a few additional heuristics to avoid in *AntNet-FA* the above-mentioned locking problem. In fact *AntNet-FA* does not suffer from the locking problem [48]. Their algorithm, *BNetL*, showed similar performance as compared to *AntNet-FA*. The authors of [74, 75] have developed a multi-agent system, which consists of static and mobile agents (ACR concept), for multiple-criteria load-balancing on a network of processors.
Oida and Sekido [146, 147] proposed the *Agent-based Routing System (ARS)* as an enhancement of *AntNet* for QoS routing. Each service class (in terms of bandwidth) has its own colony of ants. The ants move in a "virtually constrained network" and take their decisions based on the values in the routing tables and the amount of bandwidth already reserved by the ants of the colony. The ants of a colony use only those links whose available bandwidth is greater than the bandwidth constraint assigned to the colony. If the available bandwidth of a link is very small then the probability of selecting it is already made low. If an ant took too many hops or none of the out-going links has enough remaining bandwidth, then the journey of the forward ant is terminated.
Baran and Sosa [14] made the following improvements to *AntNet-FA*:

- Intelligent initialization of routing tables is done in which the entries in the routing tables are not uniformly initialized. Rather the probability values for these destinations, which happen to be neighbors of a node, are initially given a higher value.

- The algorithm explicitly sets the pheromone value of a neighbor for reaching a destination to zero if the link to the neighbor or the neighbor crashes. This pheromone value is evenly distributed among the remaining neighbors, through which a destination is still reachable. This feature makes *AntNet-FA* fault tolerant.

- *Uniform ants*, like those proposed in [191], are introduced to counter the stagnation of the entries in the routing tables of the nodes. However, we must again emphasize that *AntNet-FA* does not suffer from stagnation because of its heuristic pheromone control and privileged pheromone laying features.

- The ant agents make greedy deterministic decisions instead of random proportional ones. The policy might make ant agents infinitely loop in a cyclic path, if its greedy deterministic decisions force it to follow a cyclic path.

- The number of ants living in the network have been arbitrarily limited to four times the number of links. The authors did not provide a reason for this value. This approach might help in reducing the control overhead of ant agents when network is experiencing congestion but this will also impair the responsiveness of the algorithm to dynamic network traffic situations.

The authors of [129] replaced the stochastic decision policy of *AntNet* with a deterministic greedy policy, which does not use a queue heuristic. The authors compared this version of *AntNet* with *OSPF* on small tree, ring and star topologies, by simulating FTP traffic using TCP Tahoe. In a steady state both algorithms showed a similar performance. Also a hybrid QoS aware routing algorithm was developed [129, 130] by combining useful features of *AntNet* and *ABC*. This algorithm provides soft guarantees on two parameters: end-to-end delay and throughput. The algorithm utilizes two types of ants, one for each constraint. The *delay ants* are similar to the ant agents utilized by *AntNet*. The *throughput ants* inherit behavior of ants in *ABC*: they are artificially delayed at each node proportional to the occupied bandwidth which is measured as a local exponential average of the link utilization. Their virtual delay is a measure of the available bandwidth in the network. The experiments conducted used the same traffic patterns and topologies as mentioned before. The results revealed that the performance of the AntNet-like algorithm

is similar to *OSPF*, however, it scales better than *OSPF* under an increase in traffic load.
The authors of [106] have provided a brief overview of so called Swarm Intelligence (in practice ACO) algorithms for routing in networks. They then proposed an *Adaptive-SDR* algorithm in [105], which organizes the network into clusters by using a centralized k-mean algorithm. Once the partition process is complete then the algorithm maintains inter-clustering and intra-clustering routing tables at each node. Multiple colonies of ants are used to discover and maintain these different routing tables. In this manner the number of ants which need to be sent in the network is reduced because a node only maintains routes to the nodes within the cluster and not to all nodes in the network. However, the implementation of *AntNet* in [105] is not correct because it was asserted that *AntNet* deterministically routes data packets on highest probability paths, which is in contradiction to the stochastic spreading feature of *AntNet* [48]. The comparison of Adaptive-SDR with *AntNet* (in the erroneous version as explained above), *OSPF* and RIP showed that Adaptive-SDR achieves the best results regarding the throughput and average delay. The experiments were conducted on 16 and 48 nodes network topologies in NS-2 simulator.
Jain [99] implemented a version of *AntNet* quite similar to that reported in [129] on the NS-2 simulator. In her algorithm, data packets are deterministically forwarded in a greedy fashion to the highest probability neighbor. The experiments were run on gird networks of different sizes. The two *AntNet* algorithms exhibited the same behavior under low load scenarios, but this single-path *AntNet* algorithm [99] quickly adapted to new situations. Sim and Sun [174] proposed the MACO approach for load-balancing in connection-oriented networks, which utilizes multiple colonies each laying its own type of pheromone. An ant is expected to choose a path which has a higher pheromone concentration of its own type, and due to a concept of pheromone repulsion an ant is less likely to prefer a path with a higher concentration of pheromone, laid by ants of other colonies in order to find good (disjoint) paths. The advantages of using MACO in circuit switched routing is that it spreads data packets over multiple paths without significantly increasing the routing overhead.
Tadrus and Bai [194] developed the *QColony algorithm* for QoS routing based on principles similar to the ACR framework. QColony nodes maintain different types of pheromone tables for different types of constraints. The algorithm utilizes different classes of agents, each of which has a different priority and serves different tasks, e.g. searching a best-effort or QoS path. Each ant is routed with the pheromone table that corresponds to its type, and it updates the routing table with a weight which is dependent on its age and priority. The ants build tables in a proactive and on demand manner that are used to find feasible paths at session startup time. An application session could specify its acceptable range of bandwidth and maximum number of hops. Once the session has started, QColony also sends *soldier ants*, who provide multiple paths for load balancing and fault tolerance, and favor paths having smaller hop count values. The experiments described in [194] show that the performance of QColony is comparable to ARS and to the selective flooding algorithm [36] for small topologies and under low network traffic load, yet its performance is significantly better for large networks and heavy traffic loads. The authors of [190] and [30] have proposed simpler algorithms for QoS routing as compared with QColony.
Carrillo et al. [31] have done a preliminary study on the scalability of *AntNet*, which is merely based on a simple theoretical formulation. It is not verified through extensive experiments on large scale topologies. They have argued through their theoretical model that *AntNet* is scalable. However, the correctness of the findings, without experimental verification, is a serious shortcoming in their study. Zhong and Evans [245] did a preliminary study in which they outlined important attacks that ant agents, launched by malicious nodes, could make. They did point out that using certificates is not a feasible option for the *AntNet* algorithm because of the processing complexity of the approach. However, they did not provide a technical solution to the problem that could be implemented in the system. Their idea is to send a verification ant which follows the best path toward a destination when the goodness of a neighbor increases above a threshold. The trip time could be calculated either by dividing the round trip time by two, or by measuring the difference between the entrance time at destination and launching time at source, assuming that clocks are perfectly synchronized using GPS service. Similarly, Yang et.al [241] are the first ones who implemented *AntNet* in the application layer of the network stack and then did experiments

on a small topology of 5 nodes. The results of their experiments show the advantages of dynamic reinforcement, which is based on the trip time of ants, over constant reinforcement.
Similarly, Yang et.al [241] are the first ones who implemented *AntNet* in the application layer of the network stack and then did experiments on a small topology of 5 nodes. The results of their experiments show the advantages of dynamic reinforcement, which is based on the trip time of ants, over constant reinforcement.
We have summarized the history of ant algorithms for best effort routing in Table 2.1, and of ant algorithms for QoS routing in Table 2.2.

| Authors | Algorithm name | Year | References |
|---|---|---|---|
| Di Caro and Dorigo | AntNet,AntNet-FA | 1997 | [49, 52, 51, 55, 54, 50, 53] |
| Subramanian, Druschel, and Chen | ABC Uniform ants | 1997 | [191] |
| Heusse, Snyers, Guerin, and Kuntz | CAF | 1998 | [93] |
| van der Put and Rothkrantz | ABC-backward | 1998 | [201, 200] |
| Oida and Kataoka | DCY-AntNet,NFB-Ants | 1999 | [145] |
| Gallego-Schmid | AntNet NetMngmt | 1999 | [80] |
| Doi and Yamamura | BntNetL | 2000 | [60, 61] |
| Baran and Sosa | Improved AntNet | 2000 | [14] |
| Jain | AntNet Single-path | 2002 | [99] |
| Zhong and Evans | AntNet security | 2002 | [245] |
| Kassabalidis et al. | Adaptive-SDR | 2002 | [106, 105] |
| Yang et al. | AntNet on LAN | 2002 | [241] |

Table 2.1: Wired best-effort networks. The table is reproduced from thesis of Di Caro [48] with his kind permission

| Authors | Algorithm name | Year | References |
|---|---|---|---|
| Schoonderwoerd et al. | ABC | 1996 | [169, 168] |
| White, Pagurek, and Oppacher | ASGA | 1998 | [234, 235] |
| Di Caro and Dorigo | AntNet-FS | 1998 | [53] |
| Bonabeau et al. | ABC Smart ants | 1998 | [19] |
| Oida and Sekido | ARS | 1999 | [147, 146] |
| Di Caro and Vasilakos | AntNet+SELA | 2000 | [56] |
| Michalareas and Sacks | Multi-swarm | 2001 | [129, 130] |
| Sandalidis, Mavromoustakis, and Stavroulakis | Ant-based routing | 2001 | [163, 164] |
| Subing and Zemin | Ant-QoS | 2001 | [190] |
| Tadrus and Bai | QColony | 2003 | [194] |
| Sim and Sun | MACO | 2003 | [174] |
| Carrillo, Marzo, Fabrega ,Vila and Guadall | AntNet-QoS | 2004 | [30] |

Table 2.2: Wired QoS networks. The table is reproduced from thesis of Di Caro [48] with his kind permission

## 2.4 Evolutionary routing algorithms for fixed Networks

In this section, we provide a brief survey of routing algorithms, which have been developed on the background of *Evolutionary Algorithms (EA)*, which in turn are inspired by the evolution process in living cells. A description of the principles of evolutionary algorithms, and their application to different optimization problems, is discussed in [82, 96] and a comprehensive survey about evolutionary strategies is provided in [16].

### 2.4.1 Important elements of EA in routing

We first summarize the important features of evolutionary algorithms which have been successfully employed in routing algorithms.

#### Chromosomes

The algorithms model the solutions of a problem by encoding it as a gene, chromosome, or an individual. The algorithm randomly generates a population of the individuals by randomly altering different genes (options) in the individuals. In a routing algorithm an individual is a string that consists of a sequence of nodes.

#### Fitness

The agents are launched by the nodes, which traverse the sequence of nodes encoded in the chromosomes. Once an agent returns to its source node, the fitness of its corresponding chromosome is evaluated on the basis of the routing information collected by the agent. The fitness can be defined as a function of trip time or hop count etc. Its definition plays an important role in the performance of an algorithm.

#### Evolutionary operators

The algorithms apply selection, cross-over and mutation operators on individuals which are based on the Darwinian notion in biology. After the fitness evaluation of all individuals of the first generation, the n fittest individuals are selected for replication in the new generation. Some of them are taken as parents for a cross-over operation in which a partial solution of one individual is combined with the partial solution of another individual and vice versa. Finally, a part of solution in an individual is replaced by some another random value, and this is termed mutation. Mutation and cross-over operators provide diversity within a population. The selection operator ensures that a portion of a population consists of the fittest individuals from the previous generation. In this way an algorithm is not only able to strive for the optimal solution but also avoids stagnation. In routing algorithms, it corresponds to keeping the so far best discovered routes and also discovering/evaluating new routes, through mutation and cross-over operators. Extremely poor routes are to be extincted through continuous application of genetic operators. Evolutionary algorithms thus provide adaptation in dynamically changing environments. The approach Munetomo took in [139, 140] is promising because evolution is a distributed process in which each individual independently adapts to its environment without the need for having explicit communication with other individuals. An evolution process is also robust to changes in the environments. These features make evolutionary algorithms appealing for telecommunication networks. A detailed survey has been provided in [176]. Here we will just provide a brief summary of three state-of-the-art routing algorithms, namely *GARA* [141], *ASGA* [229] and *DGA* [120].

### 2.4.2 GARA

Munetomo [141, 139] developed the *Genetic Adaptive Routing Algorithm (GARA)*, which utilizes *path genetic operators* to identify a subset of routes which should be monitored. The fitness of a route is calculated by normalizing observed communication latencies among alternative routes to the same destination. The algorithm periodically applies path genetic operators, which remove the entries for the routes whose destinations do not frequently receive data packets and the route with a worst fitness value. As a result, routing table only contains routes to the destination where packets are frequently sent. In *GARA* path mutation and path cross-over operators are based on the topology. Through a mutation operator, a mutation node is randomly selected from a route leading to a particular destination. In the next step, a neighbor of a mutation node is randomly selected and then the source node, the selected neighbor and the destination node are connected by Dijkstra's shortest path algorithm. In the cross-over operator, a router, which exists in two

routes leading to the same destination, is selected as a cross-over point. The sub-routes after the cross-over point are then exchanged. This limits the cross-over to those routes which have at least one common node. We will illustrate both operations in the topology in Figure 2.3. Let us assume that we want to apply the mutation operator to route 9-4-2 and we select node 4 to be the mutation node. We select neighbor 3 of node 4 as a replacement. Now once we try to join node 9 with node 3 and node 3 with node 2 we get the new route 9-4-3-1-2. Let us assume that we want to apply the cross-over operator to routes 8-9-4-3 and 8-4-2-1-3. We again select node 4 as a cross-over point and achieve two new routes: 8-4-3 and 8-9-4-2-1-3. The algorithm "learns" about new routes by utilizing the above-mentioned strategies. *GARA* is a *host intelligent* routing algorithm. It requires that the sender node puts the complete route in the header of each data packet. As a source routing approach, it does not scale well for large networks where the overhead of putting the complete route into each packet will considerably increase the size of the packet, and hence cause congestion and waste of bandwidth. Therefore, the authors decided to switch to partial source routing in which only a few initial hops are put in the source header and then it switches to next hop routing [140].

### 2.4.3 ASGA and SynthECA

White et al. combined important concepts of the Ant System [66] with the ideas of genetic algorithms into the routing algorithm, *ASGA (Ant System with Genetic Algorithm)*, for circuit-switched networks [229, 234, 235]. The algorithm can be utilized for point-to-point, point to multipoint, and multi-path routing in circuit-switched networks. The algorithm follows a standard genetic algorithm in which an initial population of ant agents is generated, which are assigned random parameter values for pheromone ($\alpha_d$) and cost sensitivity ($\beta_d$) parameters for reaching destination $d$. These *explorer agents* are launched in the network and they follow their journey according to the pheromone values in the routing tables. During exploration they maintain an internal cost path variable $C_d$, instead of the trip time $t_d$. However, the authors did not clearly define $C_d$ and $l_{ij}$, the cost function of a link between node $i$ and node $j$, for circuit-switched networks. After crossing a link from $i$ to $j$, the ants update their cost variable using the formula $C_d = C_d + l_{ij}$. Once the explorers reach their destination they start their return trip and update the pheromone tables by using modified Ant System equations. After they arrive at their source node, the path found is written into a buffer, their fitness is evaluated and associated with $(\alpha_d, \beta_d)$ parameters in an another table. Finally, for the second iteration, the genetic algorithm applies selection, cross-over and mutation operators on the current generation to create a population for the second iteration. The new generation of explorer agents is again assigned $(\alpha_d, \beta_d)$ values. The genetic algorithm empowers the explorers to keep on exploring alternate paths; this feature, coupled with evaporation, privileged pheromone laying and pheromone heuristic control, enables the algorithm to avoid stagnation. In the *ASGA* algorithm, a source node decides, depending on the percentage of ants that followed the same path, whether the network or router resources should be reserved for a call. This objective is achieved by launching *allocator agents* [235], which allocate resources along the paths selected. Similarly, when a path is no longer needed then *deallocator agents* are launched. They deallocate the networks resources used in the nodes and links. The system also utilizes *evaporator agents* which circulate in the network and evaporate the pheromone concentrations that had been laid on a path to promote exploration. The authors have evaluated and described the merits of *ASGA* in [91, 233]. They conducted preliminary experiments on smaller/simple topologies, and the results show that the algorithm is dynamically able to compute shortest paths, and that the genetic adaptation of $(\alpha_d, \beta_d)$ considerably contributes to improving the performance of the algorithm. However, the considerable overhead in terms of bandwidth and processing, was not evaluated. Moreover, the performance of *ASGA* was not compared to other state-of-the-art Nature inspired routing algorithms.

Based on *ASGA* a general framework *SynthECA (Synthetic Ecology of Chemical Agents)* [230, 91, 233, 228] has been developed . A detailed review is made in [174, 230, 232, 231], and a detailed description is provided in [228]. *SynthECA* manages point-to-point, point-to-multipoint and multi-path routing like *ASGA*, with an additional feature for fault location detection and management

[230, 232].
The agents in *SynthECA* are described by a tuple, which consists of *emitters, receptors, chemistry, migration decision function (MDF)* and *memory*. The emitters associated with the agents generate chemicals, their production rate is controlled by an *Emitter Decision Function (EDF)*, and they are deposited in the ambient environment of an agent. Receptors sense chemicals in the agents' environment according to their *Receptor Decision Function (RDF)* and then take appropriate actions. The emitters and receptors are digitally encoded with 0,1 or #, where # is a wild card. For example a receptor with encoding 10## can detect chemicals 1000, 1001, 1010 and 1011 [230]. The chemistry associated with an agent is a set of rules, which determine how different types of chemicals can react together to produce different chemicals, thereby changing the local environment of an agent. The agents in *SynthECA* utilize five types of chemical reaction rules among pheromones:

1. $X \rightarrow$ 'nothing' (evaporation property)
2. $X + Y \rightarrow Y$ (survival of the fittest)
3. $X + Y \rightarrow Z$ (reproduction/stigmergy)
4. $X + Y \rightarrow X + Z$ (survival of the fittest)
5. $X + Y \rightarrow W + Z$ (reproduction/stigmergy)

Rule 1 ensures dynamism, and all other rules allow the receptor ants to communicate important network state information to other ants. Rule 2 and Rule 4 can allow high priority traffic to use the resources in the network while low priority traffic could be either discarded or diverted to another route. Rule 3 and Rule 5 can be used to communicate inhibitory or excitatory messages to other ants, which will cause these to detour from faulty portions in the network in the first case, or be attracted to good paths in the second case. The memory within an agent stores special chemicals, for which no emitter and receptor is defined, to determine next hop for the agent: MDF (the migration detection function mentioned above) determines the next hop of the agent, as a function of chemical values and link costs.
The system consists of three classes of agents: *route finding agents (RFA)*, *connection creation monitoring agents (CCMA)* and *fault detection agents (FDA)*. The route finding agents have already been described in the beginning of the section and they are explorers, allocators, deallocators and evaporators. The purpose of CCMA agents is to monitor the quality of connection parameters using special q-chemicals [230]. Finally, Fault Detection Agents observe the quality of different links by accessing q-chemicals laid by CCMA agents. Their job is to look for high q-chemical values above a threshold, and take remedial actions if needed.
*SynthECA* consists of a colony of different types of ant-like agents, which utilize chemical features along with ACO principles (Section 2.3) to solve a problem. Moreover, the agents undergo continuous evolution through an evolutionary algorithm at each node. The system is quite complex when compared with ACR (Section 2.3.5). We believe a thorough analysis about the complexity of such a system is necessary, both in terms of needed processing power and network resources, before a clear judgment about its benefits can be made, in comparison to all previously discussed approaches.

### 2.4.4 DGA

Liang et al. have studied the impact of the size of the routing table on the performance of AntNet [119]. They reduced the number of entries in the routing table of AntNet and termed the new algorithm AntNet-local, thus decreasing routing table information, hence the overhead for making routing decisions. The experiments conducted in NTTNet [119] reveal that AntNet-local has a significantly poor performance as compared with AntNet-global both in terms of throughput and delay. However, they did not discuss the trade-off between reduced overhead and quality of paths selected.

They then developed the *Distributed Genetic Algorithm (DGA)* [120] based on the concepts of *GARA* [120]. The following features of *DGA* allows for an easy application of genetic operators:

- Each node initializes a population of agents with size $size = links^2 \times c_1$ where $c_1$ is a constant. The formula is motivated by the fact that higher degree routers have to explore more links and connections between and across neighbors. Initially, only half of the agents are launched in the network.

- Each agent in *DGA* is modeled as a chromosome of integers, which represent next hop offsets. Once an agent enters a node $i$, it picks up the offset number $m_i$ from the chromosome, and then applies the formula $index_i = m_i \% N_i$ where $N_i$ is number of neighbors of node $i$. The agent then identifies a link with offset $index_i$, counting clockwise from the entering port. The link is selected as the next hop. The number of entries in the chromosomes are restricted to m (currently 6). The size of a chromosome determines how many hops, starting from the source node, an agent is allowed to take. This representation is then independent of the network connectivity, and hence simplifies the design of genetic operators. Each agent is equipped with a stack which carries the address of the visited nodes and trip time value to the visited nodes from the source node.

- An agent terminates its journey if it has visited the last entry in the chromosome. However, if $index_i$ ends up being at the same link from where the agent arrived then the next hop is selected from the remaining neighbors in a random fashion and accordingly the entry in the chromosome is altered. If no next hop can be selected because the agent has already visited all of the neighbors then the chromosome is truncated and the forward agent is converted into a backward agent.

- The backward agent only modifies the routing tables at its source node. Its fitness function is defined as $f = \frac{\sum_{k=1}^{D_i} \alpha_k^i \times t_k}{\sum_{k=1}^{D_i} t_k}$, where $D_i$ is number of explored destinations at node $i$, and $t_k$ is the trip time value for destination $k$. $\alpha_k^i$ is defined as $\alpha_k^i = \frac{m_k^i}{T_i}$, where $m_k^i$ is the total number of packets generated for node $k$ at node $i$ while $T_i$ is the total number of packets generated for all discovered destinations at node $i$. The definition assigns a high fitness value to an agent which has discovered a low latency path to a destination (where more data packets are sent). The ID of an agent, its fitness, the nodes it visited, and the trip time to the nodes are stored in a routing table (please refer to Figure 2.3).

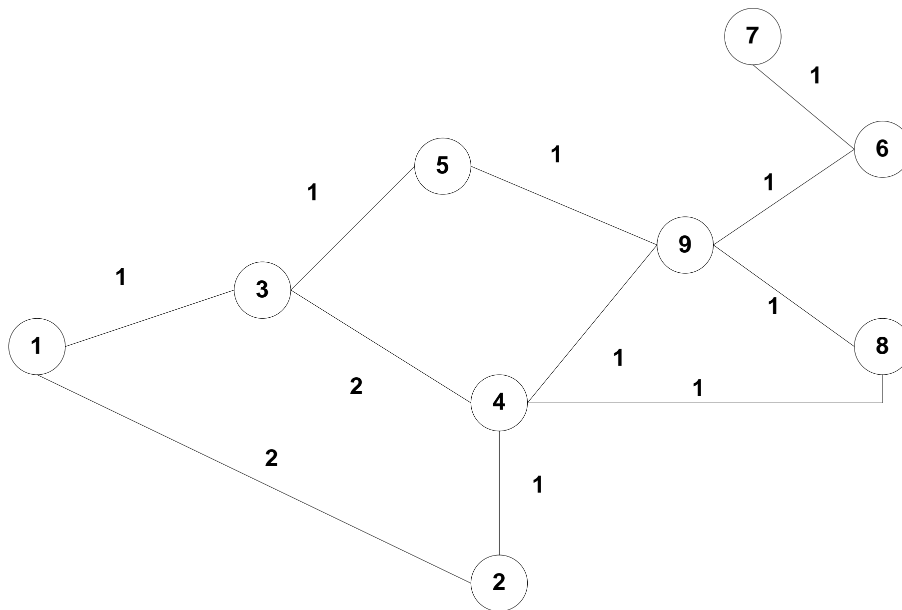
- The authors of *DGA* have introduced the concept of aging, by periodically decreasing the fitness values (f) of the agents, and at the same time increasing the trip time values ($t_k$) through formulas $f = f \times c_2$ and $t_k = \frac{t_k}{c_2}$. This will avoid stagnation in the routes. $c_2$ is set between 0.8 and 0.9.

- Once four agents return to a node then selection, cross-over and mutation operators are applied to the two best agents. Since chromosome representation is not dependent on the topology, one can simply use the traditional genetic operators. The two new agents are inserted into the node population after deleting the two worst agents from the population.

- Periodically, every 500 or 700 msec, each node passes its 3-5% best individuals to its neighbors.

- Once a node wants to forward a data packet whose destination has been discovered then it is forwarded through the agent which has the shortest trip time value to the destination. However, if the destination is still not discovered then the packet is routed through the agent which has the highest fitness value.

The authors compared their algorithm with AntNet-local on NTTNet, under a low traffic load of about 35 packets/sec. The results demonstrate that *DGA* is able to deliver more packets as compared with AntNet-local, but with a higher delay. The authors dropped data packets which




| Agent ID | Agent Fitness | node ID and Trip time (ms) |
|---|---|---|
| 85 | 0.32 | (4,10),(3,30),(1,35),(2,55),(4,65), (8,65) |
| 234 | 0.45 | (6,10),(7,20) |
| 31 | 0.66 | (4,10),(8,15),(9,20),(6,25),(7,30) |
| … | ... | ... |
| 25 | 0.81 | (5,10),(3,15),(4,25),(2,30),(1,35) |

**Routing Table at Node 9**

Figure 2.3: Routing table in DGA

followed a cyclic path, yet did not provide a proper justification for it. We provided a detailed critical review on *DGA* in [216]. *DGA* is a complex and sophisticated algorithm which launches half of the population at start up. Consequently, agents occupy approximately 50% of the bandwidth which is really not acceptable. The authors did not provide results for *OSPF* and AntNet-global. Our study [216] shows that at 35 packets/sec the performance of *DGA* is significantly inferior compared to both *OSPF* and AntNet-global. Another important drawback of *DGA* is that its control overhead increases with a decrease in traffic load. Ant agents use the same buffers as data packets, and since a next generation of agents is launched from a node once it receives four agents from the previous generation. Consequently, more agents will be traveling on the network if their trip time is small and this happens on a small topology or under low traffic loads, or both. This behavior is in contrast to the expectation: more exploration under high traffic load and low exploration under low load. Last but not least, the authors themselves were not sure about the complexity of searching and storing the population based routing tables employed by *DGA*. An exemplary routing table is shown in Figure 2.3.

## 2.5 Related work on routing algorithms for fixed networks

We now provide a very brief review of the algorithms developed by the Artificial Intelligence (AI) community and the Networking community. The motivation of doing this is to introduce state-of-the-art routing algorithms developed by these communities, which will provide the basis for cross-fertilization of ideas among all communities.

### 2.5.1 Artificial Intelligence community

The artificial intelligence community has applied Reinforcement Learning (RL) [104] algorithms, developed as a branch of machine learning, to propose routing algorithms for packet switched networks. The two well known algorithms are Q-routing [23] and PQ-routing [37], which are based on the Q-learning [212, 213] algorithm.
Q-routing employs an on-line asynchronous decision policy which is based on local information. Every router maintains Q-values, which represent the goodness of a neighbor for reaching a particular destination. The value $Q_i(j,d)$ is an estimate of the time at node $i$ that a packet will take for reaching destination $d$ via neighbor $j$. Once the neighbor $j$ receives a packet it will immediately send a feedback packet to node $i$ with a new estimate $Q'_j(d) = \min_{z \in N(j)} Q_j(z,d)$, which is the best trip time estimate held at node $j$ for destination $d$. If the feedback packet took $t_{ji}$ time, which is the sum of the propagation delay on the link and the queuing delay at node $i$ then node $i$ could revise its estimate according to the formula $\delta Q_i(j,d) = \eta(Q'_j(d) + t_{ji} - Q_i(j,d))$, where $\eta$ is the learning rate which is a standard feature of iterative algorithms and is generally set to a value which satisfies the stochastic approximation convergence [202]. The authors used a value of 0.5 in their experiments. In this way the time-to-go estimates are updated using the exponential averaging. Finally, the data packets are deterministically routed through the neighbor which has the lowest associated Q-value (highest goodness). The deterministic routing policy will keep on sending the data packets through the neighbor with the lowest Q-value until the Q-values of the other neighbors drop below the Q-value of the selected neighbor. If a neighbor recovered from a transient overload then it would never be selected as a next hop until the Q-value of all other neighbors become worse than its own. This feature provides no room for load-balancing. The authors conducted their experiments on an irregular grid of $6 \times 6$. The results show that Q-routing performs similar to the Shortest Path routing algorithm under low network loads, and performs significantly better under higher network loads. Moreover, the control overhead is directly proportional to the number of data packets switched by a node, which under high network load could be unacceptable.
Choi and Yeung [37] proposed the Predictive Q-routing algorithm known as PQ-routing, which

overcomes the above-mentioned problem. Moreover, they contemplate in [37] that Q-routing does not always converge to shortest path under low loads. In PQ-routing, they do *controlled exploration* of congested paths by occasionally sending probe packets along the paths. The probing frequency depends on the network traffic and recovery rate of a path. Q-values are updated in a similar way as in Q-routing. However, a more sophisticated routing policy is employed. The recovery rate of each neighbor is determined based on the difference in two subsequent $\delta Q$ values for each neighbor. Then, based on the recovery rate of all neighbors, existing Q-values and best estimated Q-values, the next hop leading to a particular destination is selected. The authors conducted a number of experiments on a 13 node topology, and a $6 \times 6$ irregular grid. The results demonstrate that the adaptation time of PQ-routing is significantly smaller once traffic patterns or topologies change because PQ-routing utilizes the concept of recovery rates. PQ-routing is generally better than Q-routing under both low load and varying network conditions, but its performance becomes comparable with Q-routing under high load conditions.

### 2.5.2 Networking community

The most important work in the field has been contributed by the Networking community, which also considers itself as pioneering packet-switched networks. The roots of its work go back to the development of ARPANET and a routing algorithm for it that is based on asynchronous distributed Bellman-Ford algorithm [15, 136]. The basic idea of the algorithm is that each router maintains only best known cost paths to each destination. Each router forwards its current routing table to all of its neighbors, as a vector of distances to all nodes in the network. Once the neighbors receive the estimates then they compare these estimates sent by their neighbors with their own estimates. If a neighbor's cost estimate to a destination is less than the current estimate of a node then the node accordingly updates its routing table with the new estimate. The algorithm iteratively progresses toward stability. After the first iteration, the routers know the current best path costs to all routers which lie within one hop diameter of the routers. The diameters keep on expanding by one with each iteration until each router has routing information for all nodes in the network. The algorithm, however, suffers from count-to-infinity and the looping problems [15, 136, 152]. As ARPANET grew bigger, many researchers proposed novel adaptive routing algorithms in [29, 79, 78, 109]. In 1980s ARPANET was transformed into NSFNET, which became the T1 US backbone. *OSPF*, which is a link-state routing protocol, was developed for NSFNET. The link-state routing algorithms avoid looping and count-to-infinity problems. *OSPF* is currently the state-of-the-art routing algorithm employed as an Interior Gateway Protocol (IGP). The Routing Information Protocol (RIP) [90, 123] algorithm based on the asynchronous distributed Bellman-Ford algorithm is used for routing within an AS (autonomous system) in the Internet [137]. *OSPF* stores the entire topology of a network in a weighted directed graph, in which each edge corresponds to a link, and each node corresponds to a router. The cost of a link is a function of the propagation delay of the link. However, the network administrators are also allowed to change these costs based on their on-field knowledge about network traffic loads. Each node in the *OSPF* algorithm estimates the costs of the links to its neighbors. It then encapsulates the addresses of its neighbors and the costs of the links to the neighbors in a link-state packet and broadcasts it to all of them. The neighbors, in turn, send the link-state packet to their neighbors and so on until all nodes in the network get the packet. Each node builds the network topology in a distributed fashion. Finally, each node builds a shortest path tree to all destinations by considering itself as a root. The shortest paths from the root to all nodes are calculated using the deterministic Dijkstra Algorithm [57]. The next hop on the shortest path to each destination is stored in a routing table. The interested reader can find a detailed description of this algorithm in [137] and its complete implementation in [138]. However, none of the algorithms tries to maintain multiple paths to a destination at a given node. This shortcoming results in an under-utilization of network resources which consequently results in their poor performance.

In the last decade of the previous century research started on multi-path routing algorithms. Chen, Durschel and Subramanian developed an algorithm *MP-Scout* [34, 36, 35] in which multiple paths

are maintained at a given node for each destination node. *MP-Scout* is based on the concept of backward learning for determining the loop free multi-paths. A simple version of *MP-Scout* is known as *SP-Scout* in which a destination node periodically floods scout messages. Each scout message is uniquely identified with a tuple (d, $C_d$, x) where $C_d$ is the cost to reach $d$ and x is an increasing sequence number. $C_d$ is initially set to zero. The time interval between two scout floodings is known as a broadcast interval (BI). When a node $i$ receives a scout message from its neighbor $j$ for destination $d$ then the node first updates the cost parameter of the scout to $C'_d$ where $C'_d = C_d + C_{ij}$, where $C_{ij}$ is the cost of the link from node $i$ to neighbor $j$. During the first BI, the node immediately forwards the first received scout to all neighbors of the node except from which it arrived. The node might receive more scouts from other neighbors within the same BI but they are not forwarded once a scout has already been forwarded. Consequently, it will just remember the best scout, and the neighbor from which it has been received is termed as the *designated neighbor* (next hop) for its source node. In subsequent intervals, the scouts from the designated neighbors are forwarded only. If $i$ did not receive any scout from the designated node in the last BI, then it will again flood the first scout and update its *designated neighbor* for the source node of scout message. In *MP-Scout*, the scout message is identified with a tuple (d, $C_d$, sID, pID) where pID corresponds to a path ID while sID is the scout ID, $d$ is its launching node and $C_d$ is the cost to reach $d$. The launching node modifies the pID for scout messages launched on different links. Based on the pID it can identify whether the paths through different scouts can lead to a loop. The algorithm also applies two types of thresholds: scout and data. A scouting threshold sets a limit K for best known routes to be maintained at an intermediate node. However, this may result in propagating the scouts which have traversed inferior paths. The algorithm applies the concept of data threshold in which a scout which advertises a path with a greater cost than the minimum cost path (by a given percentage) is discarded. Each data packet carries an additional path ID field in its header to enable the routers to forward it to a next hop node which lies on this path ID. *Equal-cost multi-path (ECMP)* [137] is a routing technique in which multiple packets are distributed, typically in a simple round-robin fashion, on multiple paths by assigning equal costs to them. Optimal splitting by using such ECMP is not possible as the costs are not updated and they do not model traffic loads. A better optimal splitting algorithm, *OSPF-OMP*, has been proposed in [204]. *OSPF-OMP* samples the current network traffic load via opaque LSA within an *OSPF* area. The information carried can be link loading, link capacity and a measure of packets dropped. The information contained in a LSA is used to change load splitting decisions.
Recently, Vuturkey has proposed three multi-path routing algorithms: The *Multi-path Partial Dissemination Algorithm (MPDA)* [210], *MPATH* [209, 208] and *Multi-path Distance Vector Algorithm (MDVA)* [207]. All of these algorithms make use of *loop-free invariants (LFI)* discussed in [210] to ensure loop freedom at every instance. The condition is: for each destination $d$, a node $i$ can choose a successor $n$ whose distance to $d$, as known to $i$, is less than the distance of node $i$ to node $d$. The interested reader will find a complete review of these algorithms in [211].
*MDVA* [207], as the name suggests, is a multi-path distance vector algorithm which tries to overcome count-to-infinity and the looping problems in the distance vector algorithms as discussed in the beginning of the subsection. The algorithm avoids looping by sending the cost estimates along a *directed acyclic graph (DAG)* rooted at a destination. Each node in DAG computes its cost to the destination by using the costs reported by "downstream" nodes, and it reports its costs to the "upstream" nodes. This method is called *diffusing computation* and was suggested in [58] in order to ensure that a distributed computation will always terminate if they are performed in an acyclic fashion. *MDVA* uses a RIP-like algorithm to compute the cost $D^i_d$ of reaching destination $d$ from node $i$, and $SG_d$ (DAG) which is defined by a link set consisting of successor nodes leading to destination $d$. The paths in $SG_d$ should have the *loop-freedom* and *connectivity* characteristics for efficient routing. Each node in the network maintains $D^i_d$, the successor set $S^i_d$, the feasible distance $FD^i_d$, the reported distance $RD^i_d$, and $SD^i_d$, which is the shortest possible distance through the successor set. The table also stores a set of waiting neighbors in a diffusing computation. Each node also maintains a neighbor table in which $D^i_{jd}$, the values for the distance of neighbor $j$ to $d$ as communicated by $j$, are stored. The link table stores the cost $l^i_j$ for the

adjacent link to each neighbor. At startup time, a node initializes all distances in its tables to infinity, and all successor sets to null. If a link is down or a node is unreachable then its cost is considered infinity. Nodes executing *MDVA* exchange messages of the form (type, d, c) where type could be QUERY, UPDATE and REPLY, and $d$ is destination node and c is the distance to destination node. Upon arrival of a message, a node updates its routing tables. This step is repeated if the cost or status (up/down) of an adjacent link changes. When an adjacent link becomes available then the node sends an update message of type (UPDATE, d, $RD^i_d$) for each destination $d$ over the link. When an adjacent link to a neighbor $j$ fails then the neighbor table associated with neighbor $j$ is deleted and the cost of the link is set to infinity. Similarly, when the cost of the link to $j$ ($l^i_j$ ) changes then $l^i_j$ is set to the new cost, and the vector tables for each destination are updated. A node can be either in ACTIVE or PASSIVE state with respect to a destination. Initially, all nodes are in the PASSIVE state, and as long as the link cost decreases, *MDVA* works like RIP and nodes will remain in the PASSIVE state. However, if the distance to a destination increases, either because an adjacent link cost changed or a message is received from a neighbor, then the diffusing computation (ACTIVE state), as described before, is started by sending QUERY messages to all neighbors with the shortest distance $SD^i_d$ through $S^i_d$, and then waiting for the neighbors to reply. If the increase in distance is due to a query from a successor, the neighbor is added to the list of waiting neighbors waiting for a reply. When all replies have been received via REPLY message, the node can be sure that the neighbors have incorporated the distances that the node reported, and it is now safe to migrate to the PASSIVE state. If a node in the ACTIVE state receives a QUERY message from a neighbor which is not in $S^i_d$ then a REPLY is immediately sent. However, if it is in $S^i_d$ then a test is made to verify if $SD^i_d$ increased beyond the previously reported distance. If it did not, then a reply is immediately sent. However, if it did increase then the QUERY is blocked, and the neighbor is added to the neighbors' list. The replies to such neighbors are deferred until the node is ready to go to the PASSIVE state. When all replies have been received and the distance $D^i_j$ increased again then the ACTIVE phase is extended by sending a new set of queries, otherwise the active phase ends. In case that the ACTIVE phase continues then no REPLY messages are sent to the pending queries, otherwise all replies are sent and the node enters the PASSIVE state.
*MPDA* [210, 211] is obtained by applying loop-free invariants conditions to the *PDA algorithm*, a link-state routing algorithm in which "enough" routing information in the network is propagated so that each router has *sufficient* link-state information to compute the shortest paths to all destinations. In PDA, a router communicates information to its neighbors about only those links that are part of a minimum-cost routing tree, and the router validates the link information based on the distances to the head of the links, and not on sequence numbers. In PDA, a router maintains the following information:

1. The *main topology table*, $T_i$, stores the characteristics of each link known to router $i$. Each entry in $T_i$ is a triplet (h,t,c) where h is the head, t is the tail and c is the cost of the link $h \rightarrow t$.

2. The *neighbor topology table*, $T^i_j$ , is associated with each neighbor $j$. The table stores the link-state information communicated by neighbor $j$.

3. The *distance table* stores the distances from router $i$ to each destination based on the values in $T_i$ and the distances from each neighbor $j$ to each destination based on the values in $T^i_j$ for each $j$. The distance of node $i$ to node $d$ in $T_i$ is denoted by $D^i_d$; the distance from $j$ to $d$ in $T^i_j$ is denoted by $D^i_{jd}$.

4. The *routing table* stores, for each destination $d$, the successor set $S^i_d$ and the feasible distance $FD^i_j$ which is used by *MPDA* to enforce LFI conditions.

5. The *link table* stores, for each neighbor $j$, the cost $l^i_j$ of the adjacent link to the neighbor.

The routers exchange *link-state update (LSU)* messages which contain one or more entries specifying addition, deletion or change in cost of a link in the router's main topology table $T_i$. Initially, a

router initializes all distance type variables with infinity, and node type with null. Once a router receives a LSU from its neighbors then it updates its routing tables and, based on the information, constructs a shortest path tree. A router only communicates the differences in the tree to its neighbors. When two or more neighbors report different costs of the same link then the conflict is resolved in favor of the neighbor reporting the shortest distance. In case both report the same costs then the tie is broken in favor of the lowest address. In *MPDA*, the routers operate in two states: ACTIVE and PASSIVE. If a router is in the PASSIVE state and there is no change in its topology $T_i$, then the router has nothing to report and it remains in this state. However, if the router, in the PASSIVE state, receives an event that changes its topology, the router sends those changes to its neighbors, and goes to the ACTIVE state and waits for acknowledgments ($ACKS$). Please remember that LSUs are acknowledged in MDPA and inter-neighbor synchronization spans only a single hop, unlike the synchronization in [58] which potentially spans the whole network. In the ACTIVE state the router updates only $T_j^i$ and $l_j^i$. The topology table $T_i$ is updated at the end of the ACTIVE phase, after acknowledgments from all neighbors have been received. In this way router $i$ incorporates the latest changes that occurred during the ACTIVE phase in $T_i$. If no changes occur in $T_i$ then the router does not have to report anything and goes back to the PASSIVE state, otherwise a new LSU is sent to the neighbors and the router immediately goes to the ACTIVE state again. When a router detects that an adjacent link failed, any pending ACKS from this neighbor are considered to have been received. *MPDA* could not suffer from deadlocks because all LSUs are acknowledged in a finite time.
*MPATH* [209, 208] belongs to the class of routing protocols in which distance-vectors are combined with the identity of predecessor nodes, which are just before the destination node on a shortest path. The following information is stored at each node

1. The *main distance table* contains $D_d^i$ and $p_d^i$, where $D_d^i$ is the distance of node $i$ to destination $d$ and $p_d^i$ is the predecessor to destination $d$ on a shortest path from node $i$ to node $d$. The table also stores, for each destination $d$, the successor set $S_d^i$, the feasible distance $FD_j^i$, the reported distance $RD_j^i$, and two flags *changed* and *report-it*.

2. The *main link table* $T_i$ is the node's view of the network and contains links represented by (m, n, c) where (m, n) is a link with cost c.

3. The *neighbor distance table* for neighbor $j$ contains $D_{jd}^i$ and $p_{jd}^i$ where $D_{jd}^i$ is the distance of neighbor $j$ to $d$ as communicated by $j$, and $p_{jd}^i$ is the predecessor of $d$ on a shortest path from $j$ to $d$ as notified by $j$.

4. The *neighbor link table* $T_j^i$ is the neighbor j's view of the network as known to $i$ and contains link information.

5. The *adjacency link table* stores the cost $l_j^i$ of the adjacent link to each neighbor $j$. If a link is down its cost is infinity.

Each message contains an update entry like (d, c, p), where c is the cost of the node sending the message to destination $d$ and $p$ is the predecessor node on the path to $d$. Each message carries two flags: *query* and *reply*. MAPTH is based on PATH, which essentially uses the same update procedures as PDA but differs only in the types of messages exchanged. We skip the details for the sake of brevity but the interested reader will find the details in [209, 208]. *MPATH* maintains a routing graph in the form of a link set $SG_d$ for destination $d$ which remains a directed acyclic graph (DAG) at every instant. Consequently, the shortest-path trees are also shortest path multipaths. The routers operate in two states: ACTIVE and PASSIVE. The router only switches from the PASSIVE state, which is a steady state, to the ACTIVE state once it receives a message indicating that its cost to a destination has increased above the previously reported $RD_d^i$. The node then sends an update message by setting a *query* flag in it. While in the ACTIVE state, a node is allowed to update $S_d^i$ after it received replies form all its neighbors. The router goes back to the PASSIVE state if none of the distances was increased beyond $RD_d^i$. Otherwise, the router will remain in the ACTIVE state and start the above-mentioned cycle again. When a router detects

that an adjacent link failed, an implicit reply with infinite distance to the destination is assumed. This mechanism ensures freedom from deadlocks and that all routers will switch to the PASSIVE state with correct distances to destinations. All of these algorithms follow the classic model of a network routing protocol development: they use non-intelligent link-state packets to collect the information about the costs to neighbors and then propagate them in the complete network. Consequently, they suffer from the same problems: "wrong" or "out-of-order" estimates have a global impact, and the link-state algorithm requires a global system model to execute Dijkstra's shortest path algorithm. The algorithms are complex and they slowly react to changes in the topologies.
The algorithms discussed in this chapter can be easily classified along two dimensions: route discovery and packet switching. Some algorithms discover routes in a probabilistic manner, and some in a deterministic manner and this holds for packet switching as well. Figure 2.4 classifies the representative algorithms along these lines. The classification of the routing algorithms with

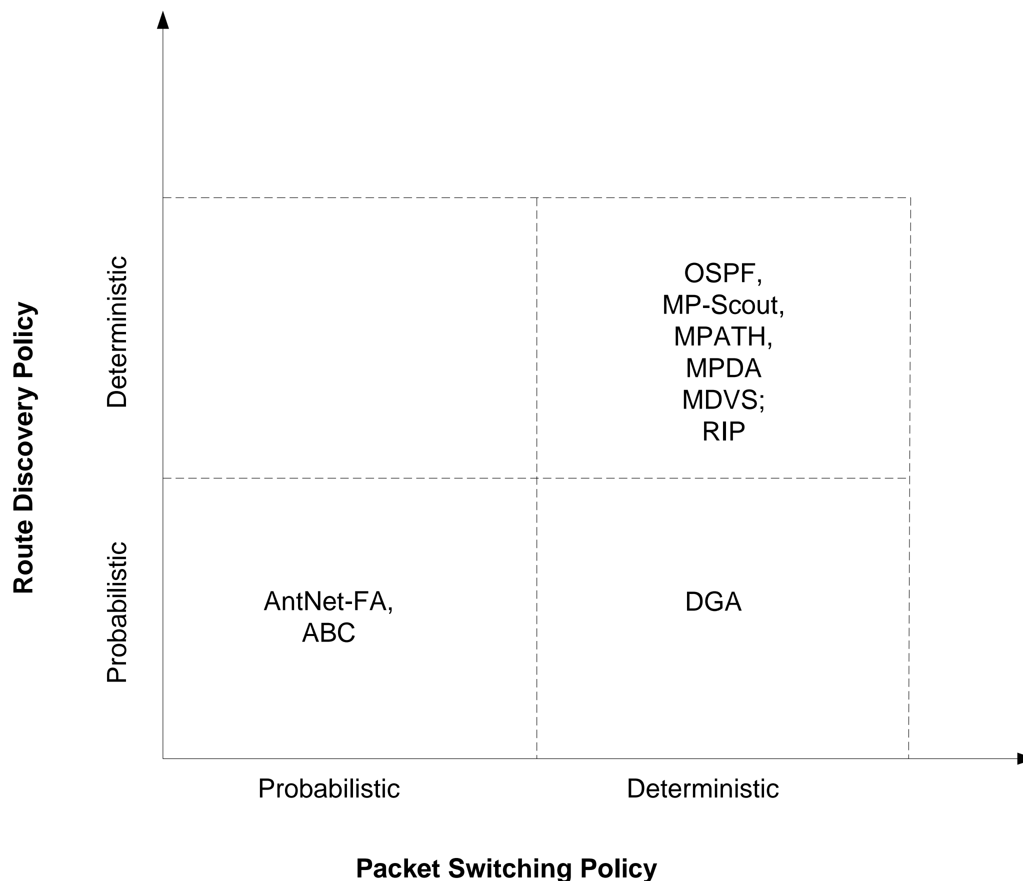



Figure 2.4: Routing classification

respect to the other design axis, introduced in Section 2.2.2, is provided in Table 2.3.

## 2.6 Summary

The efficient utilization of network resources is becoming an important issue in traffic engineering. One solution to such challenges is to design efficient, decentralized, fault-tolerant and multi-path routing algorithms which accomplish the task of routing with no access to global topological in-

| | Routing Algorithms | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Feature | AntNet-FA | ABC | DGA | BeeHive | Q-Routing | MDVA | MPDA | MPATH | OSPF |
| Static (S) vs. Dynamic (D) | D | D | D | D | D | D | D | D | D |
| Host Intelligent (HI) vs. Router Intelligent (RI) | RI | RI | RI | RI | RI | RI | RI | RI | RI |
| Single Path (SP) vs. Mulipath (M) | M | S | S | M | M | M | M | M | S |
| Constructive (Co) vs. Destructive (De) | De | De | De | Co | De | De | Co | Co | Co |
| Fault Tolerant | No | No | Yes | Yes | No | Yes | Yes | Yes | Yes |
| Global (G) vs. Local (L) | L | L | L | L | L | G | G | G | G |
| Flat (F) vs. Hierarchical (H) | F | F | F | Hybrid | F | F | F | F | F |
| Loop Freedom | No | No | No | No | No | Yes | Yes | Yes | Yes |
| Load Balancing | Yes | Yes | No | Yes | No | Yes | Yes | Yes | No |
| Stigmergy (St) vs. Direct Communication (Dc) | St | St | - | Dc | - | - | - | - | - |
| Best Effort (B) vs. QoS (Q) | B | B | B | B | B | B | B | B | B |

Table 2.3: Classification of routing algorithms for fixed networks

formation. In this chapter we have provided a comprehensive survey of state-of-the-art routing algorithms designed and developed by communities, which have different design doctrines. We believe that the survey will be instrumental in initiating an integrated approach to routing in telecommunication networks by allowing cross-fertilization of design principles from different design philosophies.

We have briefly introduced two types of Nature inspired routing algorithms: ACO inspired and Evolutionary. The agents in ACO inspired routing algorithms communicate indirectly through the environment (stigmergy) and the agents provide positive feedback to a solution by laying pheromone on the links. Moreover, they have negative feedback through evaporation and aging mechanisms which avoids stagnation. The evolutionary algorithms achieve adaptivity by applying the genetic operators of *cross-over, mutation and selection* to their population of agents. We believe that the detailed survey was instrumental in identifying the benefits and shortcomings of the existing state-of-the-art Nature inspired routing algorithms. Consequently, we were able to do a comprehensive requirements engineering for our *BeeHive* algorithm.

# 3

# From *the Wisdom of the Hive* to Routing in Telecommunication Networks

*The major contribution of the chapter is a dynamic, simple, efficient, robust, flexible and scalable multi-path routing algorithm,* BeeHive*, which is inspired from the foraging principles of honey bees. The communicative model of bees was instrumental in designing intelligent bee agents, which are suited for large and complex topologies. The results obtained from the extensive simulation experiments conclude that* bee agents *occupy smaller bandwidth and require significantly smaller processor time as compared to the agents of existing state-of-the-art algorithms. However, even with such simple agents,* BeeHive *achieves similar/better performance as compared to that of state-of-the-art routing algorithms like* AntNet.

## 3.1 Introduction

The major contribution of the work presented in this chapter is a dynamic, simple, efficient, robust, flexible and scalable multi-path routing algorithm, *BeeHive*, which has been designed based on inspirations from the wisdom of the hive. We turned these concepts into an engineering approach thus allowing us to realize the resulting algorithm in a network stack of Linux [1] The engineering approach is a result of discussions with network engineers in our system management group who have extensive experience working with Cisco routers. Consequently, we adopted only those features in the algorithm design phase which are easily realizable in a real Linux router.
We set the following challenging requirements for *BeeHive* in order to simplify its implementation in the network stack of a Linux system:

1. *BeeHive* must utilize only forward moving agents to accomplish the routing task.
2. *Bee agents* must not contain any stack for carrying out their duties.
3. The portion of bandwidth occupied by *bee agents* should be less than 1% of the bandwidth.
4. The algorithm must not maintain any statistical variables to calculate the quality of a link.
5. The time needed to process *bee agents* at a node should be kept to a minimum possible.
6. The size of a routing table in *BeeHive* must be of the same order as that of *OSPF*.
7. *BeeHive* must be able to scale to large network topologies.
8. *BeeHive* must be able to handle router/link failures in the networks.

[1] Linux was chosen because it is an open source free operating system. The availability of source code significantly helped in replacing *OSPF* with *BeeHive* in the routing framework.

9. The performance of *BeeHive* must be at least as good as that of existing state-of-the-art routing algorithms like *AntNet* and *DGA* under high traffic loads, and better than that of *OSPF* under low/static traffic loads.

10. The implementation of *BeeHive* in a network simulator must not use any simulator specific features that are not available inside the network stack of Linux kernel.

The motivation behind challenges 1 and 2 is two-fold: one, the size of an agent is not dependent on the number of hops it makes, and this results in significantly smaller waste of bandwidth of the network (challenge 3). Secondly, the time to process stack-less agent at a node turns out to be significantly smaller than in the presence of a stack. The result of meeting challenge 2 and 4 is of direct impact on meeting challenge 5. The smaller routing tables (challenge 6) provide two benefits: they occupy small memory, and secondly, they can be easily loaded into the cache of a router for doing efficient packet switching. If *BeeHive* is able to meet challenges 1 to 6 then it is expected to meet challenge 7 automatically.
Our results from extensive simulations clearly demonstrate that bandwidth which *bee agents* occupy is significantly smaller than the one used by *ant agents* in *AntNet*. The time to process agents in *BeeHive* is also significantly smaller than that of *ant agents* in *AntNet*. Also, the size of the routing tables is significantly smaller than in *AntNet*. However, even with a simpler agent and learning model, *BeeHive* achieves similar or better performance as compared with *AntNet*. The advantages of *BeeHive* over *AntNet* become more obvious in larger topologies.

### 3.1.1 Organization of the chapter

We will provide a short review of working principles of a honey bee colony in Section 3.2 which will help the reader in comprehending the bee behavior and how it differs from ant behavior. This will assist the reader in understanding the mapping of concepts from Nature to Networks as described in Section 3.3. We will define our agent model in Section 3.4, based on this we will introduce our algorithm in Section 3.5. In Section 3.6 we will introduce our performance evaluation framework. This constitutes another important contribution of the work presented in this chapter. It is meant as a basis for an unbiased evaluation of the routing algorithms. We will provide a brief description of existing state-of-the-art routing algorithms, with which *BeeHive* is compared, in Section 3.7 emphasizing the novel direction of our work. Section 3.8 will describe our simulation environment for the extensive experiments reported in Section 3.9. Finally, we will provide a short summary that concludes the chapter.

## 3.2 An agent-based investigation of a honey bee colony

In this section we briefly outline the organizational principles of a honey bee colony that will enable computer scientists to develop agent-based algorithms for different optimization and real world problems. A honey bee colony pragmatically solves the most interesting multi-objective optimization problem: how to allocate resources to different tasks under a constantly changing operating environment so that the colony maximizes its profit. The interested reader can find details in [170, 206]. This is the same problem that many researchers try to solve in the field of multi-objective optimization [183, 46, 26] under dynamic and time-varying environments.

### 3.2.1 Labor management

A honey bee colony is organized with morphologically uniform individuals but with different temporary specializations. The benefit of this approach is that it enhances a colony's flexibility to adapt its response according to an ever changing environment while at the same time doing the tasks with an acceptable level of efficiency. For example, a nectar forager is able to extensively forage at a discovered flower site as she does not waste time in storing the nectar inside the hive.

Moreover, a bee colony is able to adapt the activity level of its specialists according to the group's needs. For example, nectar foragers may become pollen foragers if the amount of protein that they receive from nurse bees falls below a threshold level, or nurse bees might take the role of food-storer bees if the rate of processing nectar is slower than the rate of collecting nectar (foragers indicate this through a tremble dance).

### 3.2.2 The communication network of a honey bee colony

A honey bee colony utilizes a hybrid communication network that consists of *signals* and *cues* for information exchange among its members. *Signals are information-bearing actions or structures that have been shaped by the natural selection to convey specific information in a unique wise manner. Cues are variables that likewise convey information about the state of the colony but have not been modeled by the natural selection to convey that information* [175]. Signals enable direct communication among the members of a honey bee colony via waggle dance, tremble dance and shaking signal while cues enable indirect communication among the members through the environment shared by them. Both mechanisms provide an efficient information exchange mechanism that empowers the members to mostly communicate indirectly (group-to-one paradigm), and when required, directly using the one-to-one paradigm.
A good example of a cue is the search time for finding a food-storer bee that a forager experiences once she wants to unload her nectar. A nectar forager uses this cue to get an estimate of both nectar collecting and nectar processing rates of the colony; higher search time to find a food-storer bee is an indicator to the forager that the rate of processing nectar is slower than the rate of collecting nectar. Consequently, she will decide to perform a tremble dance instead of a waggle dance. The forager, by doing a tremble dance, will achieve two objectives: one, she will recruit more food-storer bees to increase the rate of processing nectar, and two, she will request other foragers, on the dance floor, to stop performing waggle dances, as a result, the rate of collecting nectar will be decreased.

### 3.2.3 Reinforcement learning

A colony experiences a strong fluctuation in the external supply or internal demand (or both) for its commodities: nectar, pollen and water. The feedback signals, both negative and positive, are important to regulate their amount so that the colony has sufficient stockpile of each of these commodities. This is achieved by recruiting unemployed foragers for finding good supply sites through waggle dances. A forager decides to dance only if the quality of the food site, visited by her, is above a certain threshold or it experiences a very small search time to locate a food-storer bee for its commodity (a cue that the colony needs the commodity). By keeping the search time within certain thresholds, the honey bee colony reinforces the foraging labor at a site in times of need and vice versa. A stochastic model for the foraging behavior has been presented in [171]. Sumpter used this basic model to come up with an agent-based model in [192]. Sumpter's model provides a solid foundation for developing an agent-based Reinforcement Learning algorithm [104].

### 3.2.4 Distributed coordination and planning

A honey bee colony achieves coordination among its thousands of members without any central authority. The colony does not have a hierarchy where some individuals require information and then allocate tasks to different members and monitor them. Each individual decides to do a job depending upon the need of the colony that it estimates using the above-mentioned communication facilities and measures.

### 3.2.5 Energy efficient foraging

The foragers tend to optimize the *energetic efficiency* of foraging at a flower site. For example, if during an average foraging trip a forager collects G units of energy, expends C units of energy, and

spends time T, then *energetic efficiency* could be defined as $(G - C)/C$. Consequently, the sites that are in the neighborhood of the hive get preference than the sites far away from it [170]. That is why von Frisch believed that (short distance) foragers which return from nearby sites perform round dances while other (long distance) foragers perform waggle dances [206]. This principle enables a colony to collect the commodities at an optimum expenditure of energy.

### 3.2.6 Stochastic selection of flower sites

The unemployed foragers on the dance floor observe, at the maximum, two or three dancers and then decide to choose one among them at random. They do not broadly survey the dance floor to identify the best flower site. This concept is contradictory to human society where well informed customers are crucial to proper functioning of competitive markets. According to Seeley, this stems from the fact that whereas the individual human tries to maximize her or his own profit, the individual forager seeks to maximize her colony's profits [170]. This "sacrifice for group principle" enables a colony to distribute its forager force over different flower sites rather than allocating it to the best site only. Such a policy results in quick reallocation of foragers, which were foraging at the best site, to other discovered sites, once the best flower site is about to fade away.

### 3.2.7 Group organization

The employed foragers that collect nectar from the same type of flowers recognize one another in the hive through the flower fragrance that clings to their body. Only group companions respond to the dances and they show no interest in the dances performed by the foragers of other groups, which have been foraging at other types of flowers. However, the employed foragers might switch into another group if the quality of their flower site degrades to an extent where it is no more profitable to continue foraging at this site [206].
In the next section we will elaborate the most important step of our engineering approach: the mapping of concepts in a honey bee colony, discussed in the previous section, to an operating environment of real packet switching networks. This step will help the reader in understanding our algorithm described in Section 3.4.

## 3.3 BeeHive: The mapping of concepts from Nature to networks

In this section we briefly illustrate the mapping of concepts from Nature to Networks, one of the most important steps in *Natural Engineering*, that will simplify for the reader to trace back the origin of important features of our *BeeHive* algorithm within the principles of a honey bee colony.

1. We could consider each node in the network as a hive that consists of *bee agents*. Each node periodically launches its *bee agents* to explore the network and collect the routing information that provides the nodes visited with the partial information on the state of the network. These *bee agents* can be considered as scouts that explore and evaluate the quality of multiple paths between their launching node and the nodes that they visit.

2. *Bee agents* provide to the nodes which they visit, with the information on the propagation delay and queuing delay of the paths they explored. These lead to their launching node from the visited nodes. One could consider the propagation delay as a distance information, and the queuing delay as a direction information (please remember bee scouts also provide these parameters in their dances): this reasoning is justified because a data packet is only diverted from the shortest path to other alternate paths when large queuing delays exist on the shortest path.

3. A *bee agent* decides to provide its path information only if the quality of the path traversed is above a threshold. The threshold is dependent on the number of hops that a *bee agent* is allowed to take. Moreover, the agents model the quality of a path as a function of the propagation delay and the queuing delay of the path; lower values of the parameters result in higher values for the quality parameter.

4. The majority of the *bee agents* in the *BeeHive* algorithm explore the network in the vicinity of their launching node and very few explore distant part of the network. The idea is borrowed from honey bee colony resulting in not only reducing the overhead of collecting the routing information but also helping in maintaining smaller/local routing tables.

5. We consider a routing table as a dance floor where the *bee agents* provide the information about the quality of the paths they traversed. The routing table is used for information exchange among *bee agents*, launched from the same node but arriving at an intermediate node via different neighbors. This information exchange helps in evaluating the overall quality of a node (as it has multiple pathways to a destination) for reaching a certain destination.

6. A nectar forager exploits the flower sites according to their quality while the distance and direction to the sites is communicated to it through waggle dances performed by fellow foragers on the dance floor. In our algorithm, we map the quality of paths onto the quality of nodes for utilizing the bee principle. Consequently, we formulate the quality of a node, for reaching a destination, as a function of proportional quality of only those neighbors that possibly lie in the path toward the destination.

7. We interpret data packets as foragers. Once they arrive at a node, they access the information in the routing tables, stored by *bee agents*, about the quality of different neighbors of the node for reaching their destinations. They select the next neighbor toward the destination in a stochastic manner depending upon its goodness. As a result, not all packets follow the best paths. This will help in *maximizing the system performance although a data packet may not follow the best path*, a concept directly borrowed from a principle of bee behavior: *a bee could only maximize her colony's profit if she refrains from broadly monitoring the dance floor to identify the single most desirable food* [170] (see Section 3.2).

Now we are in a position to introduce our *bee agent model* and *BeeHive* algorithm in the following sections.

## 3.4 The *bee agent* model

Our *bee agent* model consists of two types of agents: *short distance bee agents* and *long distance bee agents*. Both agents have the same responsibility: exploring the network and evaluating the quality of the paths that they traverse. They only differ in the distance (hops) that they are allowed to take starting from their launching node. *Short distance bee agents* collect and disseminate routing information in the neighborhood of their source node (up to a specific number of hops) while *long distance bee agents* collect and disseminate routing information typically to all nodes of a network. This helps in collecting the routing information as quickly as possible with a minimum processing and bandwidth overhead.
The *bee agents* that are launched from the same node form an *affinity group* in which they show interest in each others information. Once the *bee agents* of the same group arrive at the same node, but via different neighbors of the node, they access the routing information, collected by their fellow *bee agents* in the group, in the routing table. They will decide to discontinue their exploration of the network after storing their information in the routing table, if one of their members has already arrived at the node. The communication model among *bee agents* is realized as a *blackboard system* [144].

**Definition 2 (Foraging region)** *The network is organized into fixed partitions called* foraging regions*. A partition results from particularities of the network topology and the number of hops that a* short distance bee agent *is allowed to traverse. Each* foraging region *has one representative node. Currently the lowest IP address node in a* foraging region *is elected as the representative node. If this node crashes then the next higher IP address node takes over the job.*

**Definition 3 (Foraging zone)** *A* foraging zone $FZ_i$ *of a node i consists of all nodes from whom* short distance bee agents *can reach this node. A* foraging zone *may span over multiple* foraging regions*.*

The basic motivation behind the two definitions above is to combine the benefits of a flat routing scheme, in which all routers are equivalent, with a hierarchical (cluster) routing scheme, in which cluster heads (or *representative nodes*) have more functions to do than ordinary routers in the cluster. In our hybrid scheme, each node maintains current routing information for reaching nodes within its *foraging zone* and for reaching the *representative nodes* of *foraging regions*. This mechanism enables a node to route a data packet whose destination is beyond the *foraging zone* of the given node, along a path toward the *representative node* of the *foraging region* containing the destination node. Consequently, our algorithm requires routing tables whose size is of the same order as that of *OSPF*, yet a *representative node* has no special management functions except launching *long distance bee agents*. Please remember that in a hierarchical (cluster) routing scheme, a packet whose destination is outside the current cluster, is sent to the *cluster head*. The cluster head forwards it to the cluster head of the cluster which contains the destination node. Finally the cluster head of the cluster containing the destination node forwards it to the destination node. The concept of *foraging region* and *foraging zone* provides the benefit of smaller routing tables but without the overhead of routing through cluster heads.
Researchers have proposed a number of algorithms for partitioning a network into clusters [128]. The basic feature of algorithms is that a cluster should be formed in such a manner that the cluster head (or representative node) should be at the centroid of the cluster. We deliberately did not use the concept of centroid because we wanted to investigate the performance of *BeeHive* without making any optimization to the clustering algorithm. Nevertheless, we would like to mention that choosing a representative node based on the centroid concept did not significantly improve the performance of *BeeHive*. However, this minor improvement in the performance came at a greater cluster management overhead. Therefore, we did not incorporate any optimizations in the *foraging region* formation algorithm. Our results from the extensive experiments clearly demonstrate that *BeeHive* is able to provide similar/better performance as compared with existing state-of-the-art routing algorithms without any such optimizations, which clearly is a proof of its robustness. Informally, the *BeeHive* algorithm and its main characteristics can be summarized as follows:

1. All nodes start the *foraging region* formation process during the start-up phase. They try to form a *foraging region* with the same address as their own address and consider themselves to be the *representative node* of the *foraging region*. Then they launch the first generation of *short distance bee agents* to propagate their nomination in their neighborhood.

2. If a node receives a *short distance bee agent* from a node whose *representative node's* address is smaller than that of the receiving node, then it discontinues its efforts to be a *representative node* and rather it joins the *foraging region* of the *representative node* with the smaller address.

3. If a node later on learns that its *representative node* has joined another *foraging region*, as indicated by the *short distance bee agents* of the *representative node*, then the node starts the same election process as explained in Step 1.

4. The nodes keep on launching the next generations of *short distance bee agents* by following Steps 1, 2 and 3 until the network is divided into *foraging regions* and *foraging zones*. Finally, each node informs all other nodes in the network to which *foraging region* it belongs. This

step is repeated only if *foraging regions* are reshaped because of links/nodes failures in the network.

5. At the end of Step 4 the algorithm enters into a "normal" phase in which each non-representative node periodically sends a *short distance bee agent*, by broadcasting replicas of it to each neighbor site.

6. When a replica of a particular *bee agent* arrives at a site it updates the routing information there, and the replica will be flooded again, however, it will not be sent to the neighbor from where it arrived. This process continues until the life time of the agent has expired, or if a replica of this *bee agent* had been received already at a site, in which case the new replica will be killed.

7. Representative nodes launch only *long distance bee agents* that would be received by the neighbors and propagated as in 6. However, their life time (number of hops) is limited by the *long distance limit*.

8. The idea is that each agent while traveling, collects and carries path information, and that it leaves, at each node visited, the trip time estimate for reaching its source node from this node over the incoming link. *Bee agents* use priority queues for quick dissemination of routing information.

9. Thus each node maintains current routing information for reaching nodes within its *foraging zone* and for reaching the *representative nodes* of *foraging regions*. This mechanism enables a node, as explained before, to route a data packet (whose destination is beyond the *foraging zone* of the given node) along a path toward the *representative node* of the *foraging region* containing the destination node.

10. The next hop for a data packet is selected in a stochastic fashion according to the quality measure of the neighbors. The motivation for this routing policy has already been explained in Section 3.3. Please note that the routing algorithms currently employed in the Internet always choose a next hop on the shortest path [152].

Figure 3.1 provides an exemplary working of the flooding algorithm. *Short distance bee agents* can travel up to 3 hops in this example. Each replica of the shown *bee agent* (launched by Node 10) is specified with a different trail to identify its path unambiguously. The numbers on the paths show their costs. The flooding algorithm is a variant of breadth first search algorithm. The network is partitioned into two *foraging regions* 0 and 8 by following the above-mentioned Steps 1,2,3 and 4. The *foraging zone* of Node 10, which spans over both *foraging regions*, consists of Nodes 2,3,4,5,6,7,8,9,11.

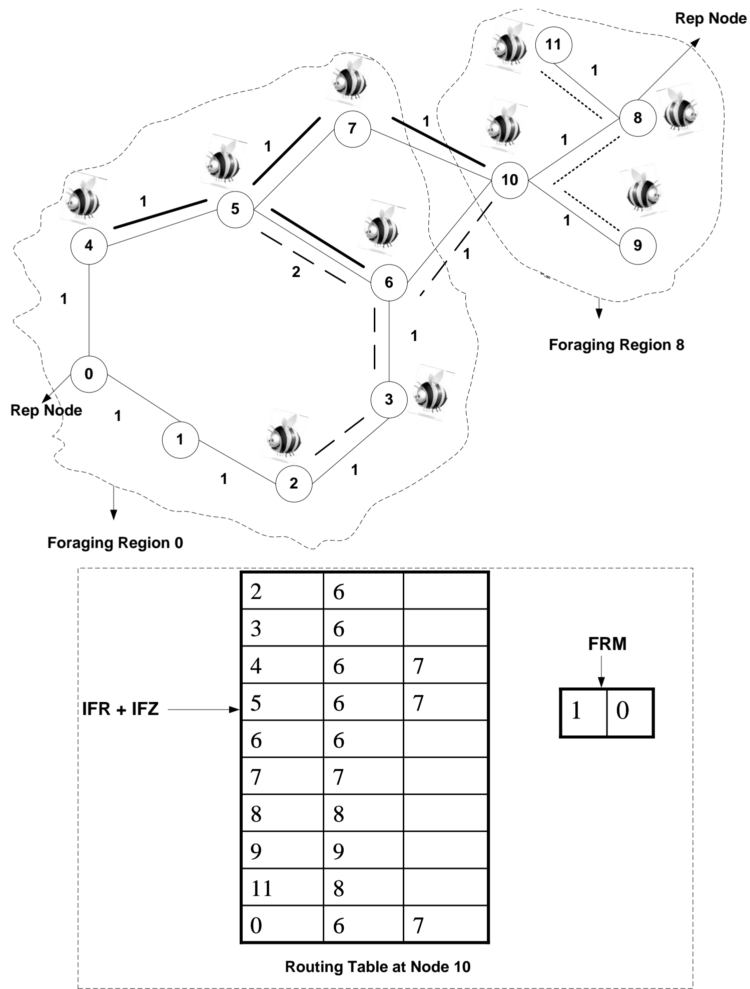
### 3.4.1 Estimation model of agents

We now briefly explain the estimation model used by *bee agents* to estimate the trip time required by a data packet to reach their launching node from the current node. In *BeeHive*, the delay $t_{in}$ that a data packet will experience in reaching a neighbor $n$ from node $i$ is modeled as follows

$$t_{in} = q_{in} + tx_{in} + pd_{in} + pr_i + pr_n \tag{3.1}$$

where $q_{in}$ is the queuing delay for neighbor $n$ at node $i$, $tx_{in}$ and $pd_{in}$ are transmission delay and propagation delay of the link between node $i$ and neighbor $n$, respectively, and $pr_i$, $pr_n$ are protocol processing delays for a packet at node $i$ and node $n$, respectively. The processing delay, generally speaking, is negligible as compared to the sum of queuing, transmission and propagation delays. Hence equation (3.1) can be rewritten as

$$t_{in} \approx q_{in} + tx_{in} + pd_{in} \tag{3.2}$$




| | | |
|---|---|---|
| 2 | 6 | |
| 3 | 6 | |
| 4 | 6 | 7 |
| 5 | 6 | 7 |
| 6 | 6 | |
| 7 | 7 | |
| 8 | 8 | |
| 9 | 9 | |
| 11 | 8 | |
| 0 | 6 | 7 |

Figure 3.1: Bee agents flooding algorithm

In equation (3.2), $tx_{in}$ models the delay experienced because of the bandwidth of the link while $pd_{in}$ models the delay that a packet experiences while traveling between two nodes on a link. $tx_{in}$ is dependent on the size of the packet, and the bandwidth of the link between node $i$ and node $n$, and $pd_{in}$ is dependent on the distance between node $i$ and node $n$. For a particular link, both of these delays do not change with the traffic loads. However, $q_{in}$ is directly dependent on the size of queue, and this in turn depends on the traffic loads. We approximate $q_{in}$ as

$$q_{in} \approx \frac{ql_{in}}{b_{in}} \tag{3.3}$$

where $ql_{in}$ is the size of the queue (in bits) for neighbor $n$ at node $i$, and $b_{in}$ is the bandwidth of the link between node $i$ and node $n$. During initialization phase, we approximate the bandwidth and propagation delays of all the links of a node by transmitting hello packets. The *bee agents* estimate the queuing delay for a link by observing the size of the queue ($ql_{in}$) and using equation (3.3). Finally, by adding the transmission and propagation delay of the link to it, they approximate the delay that a packet will experience on the link for reaching neighbor $n$. Ultimately they update the trip time $t_{is}$ that a packet will take in reaching their source node $s$ from current node $i$

$$t_{is} \approx t_{in} + t_{ns} \tag{3.4}$$

### 3.4.2 Goodness Of a neighbor

The goodness of a neighbor $j$ of node $l$ ($l$ has $N$ neighbors) for reaching a destination $d$ is $g_{jd}$ and defined as follows

$$g_{jd} = \frac{\frac{1}{p_{jd}}(e^{-\frac{q_{jd}}{p_{jd}}}) + \frac{1}{q_{jd}}(1 - e^{-\frac{q_{jd}}{p_{jd}}})}{\sum_{k=1}^{N}(\frac{1}{p_{kd}}(e^{-\frac{q_{kd}}{p_{kd}}}) + \frac{1}{q_{kd}}(1 - e^{\frac{q_{kd}}{p_{kd}}}))} \tag{3.5}$$

The fundamental motivation behind equation (3.5) is to approximate the behavior of the real network. When the network is experiencing a heavy network traffic load then the queuing delay plays the primary role in the delay of a link. In this case it is trivial to say that $q_{jd} \gg p_{jd}$ and we can see from equation (3.5) that $g_{jd} \approx \frac{\frac{1}{q_{jd}}}{\sum_{k=1}^{N}\frac{1}{q_{kd}}}$. When the network is experiencing low network traffic then the propagation delay plays an important role in defining the delay of a link. As $q_{jd} \ll p_{jd}$, from equation (3.5) we get $g_{jd} \approx \frac{\frac{1}{p_{jd}}}{\sum_{k=1}^{N}\frac{1}{p_{kd}}}$.
The plot of equation 3.5 is shown in Figure 3.2(a). Once we started developing our engineering model of *BeeHive* inside the network stack of Linux operating system as discussed in Section 1.3, we realized that the kernel library does not support mathematical functions like exponentials, sine, square root and cosine etc. We implemented the exponential function ourselves in the Linux kernel, but we observed that the processing complexity of the exponentials is an order of magnitude higher than one of simple arithmetic functions. This motivated us to look for other forms of modeling the quality function represented in equation (3.5). The basic challenge was that it should capture the network behavior as discussed in the previous paragraph yet it should contain no exponentials. The reason for having no complex mathematical functions like exponentials, sine, cosine, square root etc., is that the quality calculation considerably increases the processing overhead of a routing algorithm, therefore, its processing complexity should be as small as possible. We tried different options and the following form gave similar results as that of the form discussed in equation (3.5). The graphical representation of equation 3.6 is shown in Figure 3.2(b).

$$g_{jd} = \frac{\frac{1}{p_{jd}+q_{jd}}}{\sum_{k=1}^{N}(\frac{1}{p_{kd}+q_{kd}})} \tag{3.6}$$

We then used our profiling framework (it will be introduced in Section 3.6) to measure the processing complexity (in cycles) of both forms. Table 3.1 summarizes the results. The results are

a summary of ten independent runs and in each run the goodness forms have been evaluated 10 million times to get a more representative value. One can easily see that the new form is approximately 10 times faster to compute than the one with exponentials. The form with the exponentials takes on the average 1100 cycles as compared to 105 cycles taken by non-exponential version. However, this simple form, as the results of our extensive experiments in Section 3.9 demonstrate, has not resulted in any performance degradation as compared to that of *BeeHive* algorithm presented in [221].

| Equations | Max | Min | Av |
|---|---|---|---|
| 3.5 | 1109 | 1093 | 1099 |
| 3.6 | 118 | 96 | 105 |

Table 3.1: Processing complexity of different forms

### 3.4.3 Communication paradigm of agents

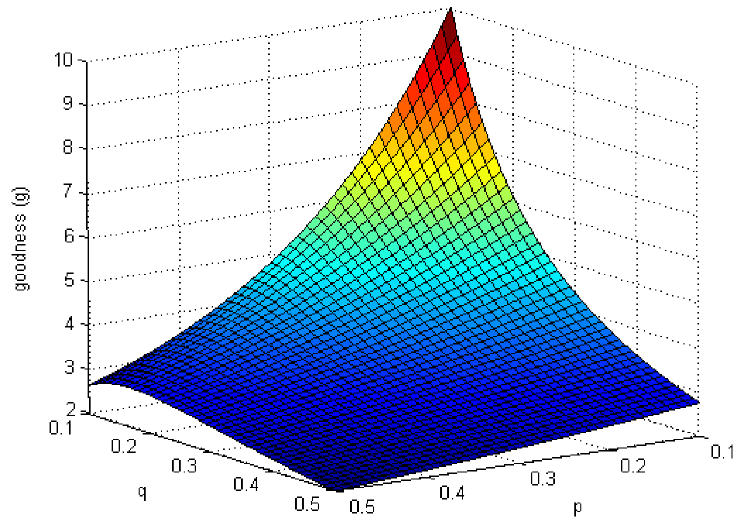
The forager bees in Nature try to exploit different food sources based on their quality, and the distance to the flower site is communicated via a waggle dance. In order to use this bee model, we need to find a mathematical model which assigns an overall quality to neighbor nodes for reaching a destination. The overall quality of a neighbor node can be further represented as a function of delays (propagation, transmission and queuing) of different paths leading from this node towards the destination. Such a model is realized with the help of a communication paradigm among different replicas of the same agent, which are launched from the same source node. The communication model is explained in Figure 3.3 in which three paths exist between $k$ and $s$. Node $s$ launches three replicas of the same agent on three paths and they arrive through different paths at node $k$. Each replica uses the estimation model described above to estimate the queuing delay and the propagation delay. The replica that arrived earlier is allowed to continue its exploration further while other replicas are killed. However, the other replicas do communicate their estimates to the replica, which is allowed to continue the exploration. Using the communication paradigm the replica calculates $p_{ks}$ and $q_{ks}$, which incorporate the estimate of all replicas proportional to the quality of the path which they traversed. Once this replica continues its exploration of the network then it tells the other nodes about existence of a path from $k$ to $s$ through which a packet could reach $s$ with a propagation delay of $p_{ks}$ and queuing delay of $q_{ks}$. The other nodes forward data packets to node $k$ based on the quality which is a function of $p_{ks}$ and $q_{ks}$. Once the data packet is at node $k$ it can take any one of three paths based on the their quality which is calculated based on the delay estimates of *bee agents*.

### 3.4.4 Packet switching algorithm

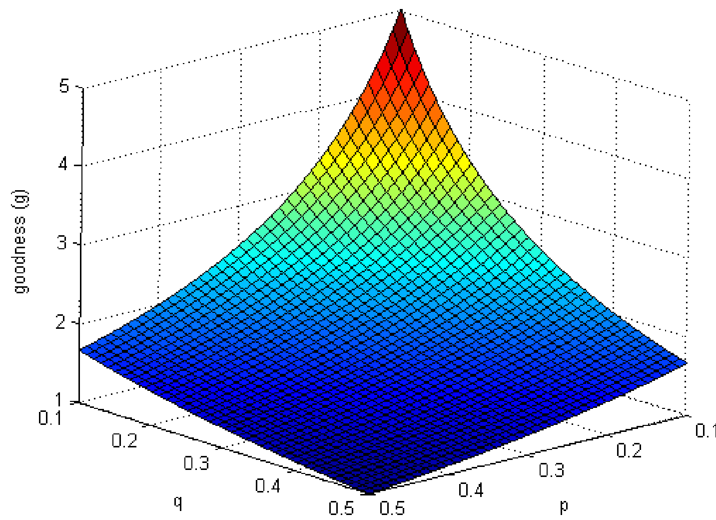
We use *stochastic sampling with replacement* [82] for selecting a neighbor as a next hop towards a particular destination. This principle ensures that a neighbor $j$ with goodness $g_{jd}$ will be selected as the next hop with at least the probability $\frac{g_{jd}}{\sum_{k=1}^{N} g_{kd}}$ or more formally , the probability of taking $j$ as a next hop towards destination $d$ at node $i$ is $\phi^i_{jd}$ and can be mathematically represented as

$$\phi^i_{jd} = \frac{g_{jd}}{\sum_{k=1}^{N} g_{kd}} \tag{3.7}$$

We did not use any rescaling of the probabilities, as done by the authors of *AntNet* because this would increase the processing complexity of doing packet switching. Please remember that any processing during the packet switching also lies on the critical path of a routing algorithm and hence has to be as efficient as possible. Our experiments suggest that the performance of *BeeHive* is same or better than the existing state-of-the-art routing algorithms even without rescaling the probabilities. This shows that trade-off between the quality of a path decision and its computing time as utilized in *BeeHive* does not harm its performance.




(a) $g = \frac{1}{\frac{1}{p}(e^{-\frac{q}{p}})+\frac{1}{q}(1-e^{-\frac{q}{p}})}$




(b) $g = \frac{1}{p+q}$

Figure 3.2: Goodness of a neighbor (different options)

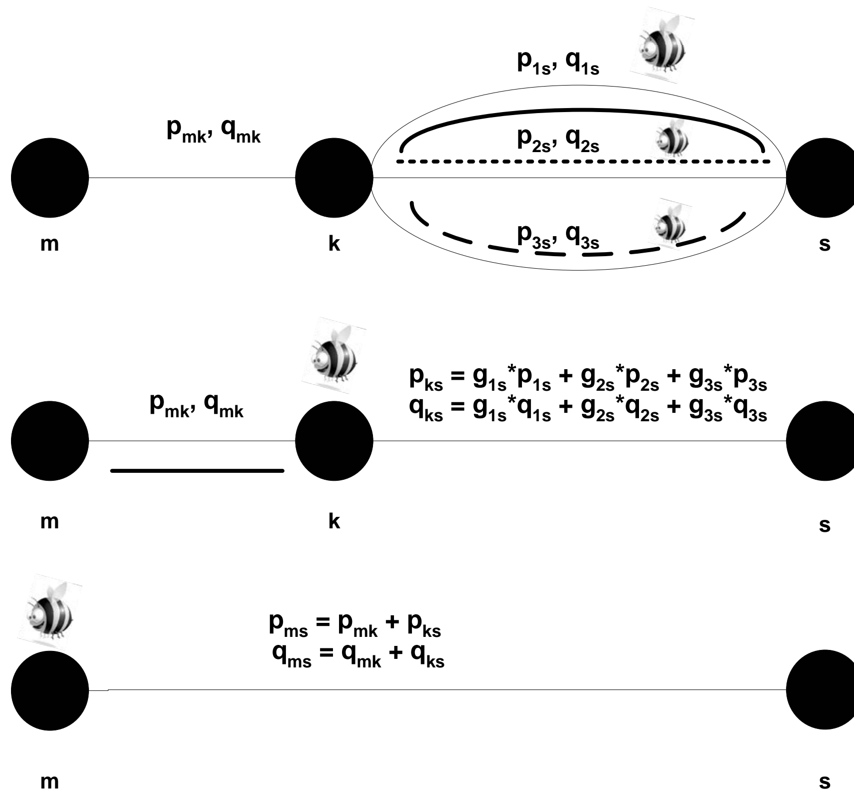



Figure 3.3: Communication paradigm of *bee agents*

## 3.5 BeeHive algorithm

In *BeeHive*, each node $i$ maintains three types of routing tables: *Intra Foraging Zone* (IFZ), *Inter Foraging Region* (IFR) and *Foraging Region Membership* (FRM). The *Intra Foraging Zone* routing table $R_i$ is organized as a matrix of size $(|D(i)| \times |N(i)|)$, where $D(i)$ is the set of destinations in the *foraging zone* of node $i$ and $N(i)$ is the set of only those neighbors of $i$ which lie on a path towards the destination. Each entry $P_{jd}$ is a pair of queuing delay and propagation delay $(p_{jd}, q_{jd})$ that a packet will experience in reaching destination $d$ via neighbor $j$. Table 3.2 shows an example of $R_i$. In the *Inter Foraging Region* routing table, the queuing delay and propagation delay values for reaching the *representative node* of each *foraging region* through the neighbors of a node are stored. The structure of the *Inter Foraging Region* routing table is similar to the one shown in Table 3.2 where the destination entry is replaced by the representative node of the region containing the destination node. The *Foraging Region Membership* routing table provides the mapping of known destinations to the *representative node* of their *foraging region*. In this way we eliminate the need to maintain O($N \times D$) (where D is total number of nodes in a network) entries in a routing table and save a considerable amount of router memory needed to store this routing table. Please have a look at Figure 3.1. The total number of entries in all routing tables maintained by Node 10 using *BeeHive* algorithm are 14. Algorithm 1 and Algorithm 2 describe the important features of the *BeeHive* algorithm.

| $R_i$ | $D_1(i)$ | $D_2(i)$ | $\ldots$ | $D_d(i)$ |
|---|---|---|---|---|
| $N_1(i)$ | $(p_{11}, q_{11})$ | $(p_{12}, q_{12})$ | $\cdots$ | $(p_{1d}, q_{1d})$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $N_n(i)$ | $(p_{n1}, q_{n1})$ | $(p_{n2}, q_{n2})$ | $\cdots$ | $(p_{nd}, q_{nd})$ |

Table 3.2: *Intra foraging zone* routing table

**Algorithm 1** *Bee launching and processing algorithms*

**procedure** launchBeeAgents($s$,$n_i$,n)
  **if** t % $\Delta t = 0$ or $n_i$ % Packet_Limit = 0 **then**
    **if** $s$ is representative node of foraging region **then**
      $h_s \leftarrow$ Long_Limit, {$b_s^v$ is a *long distance bee agent*}
    **else**
      $h_s \leftarrow$ Short_Limit, {$b_s^v$ is a *short distance bee agent*}
    **end if**
    **if** $v = 0$ **then**
      $Fr_s \leftarrow s$ {claim to be the representative node}
    **end if**
    **for** $x \leftarrow 1$ to $N$ **do**
      create a replica $b_s^{xv}$ of bee agent $b_s^v$
      find address of neighbor at index $x$
      launch replica $b_s^{xv}$ to neighbor at index $x$
      $x$++
    **end for**
  **end if**
**end procedure**

**procedure** manageRegions($s$,$Fr_s$,hops,$Fr_i$)
  **if** $Fr_s =$ s && hops $< h_s$ **then**
    **if** $Fr_s < Fr_i$ **then**
      $Fr_i \leftarrow Fr_s$ {withdraw in favor of lower IP node as a representative node}
      {If the current representative node of $i$ has joined another foraging region}
    **else if** $Fr_i = Fr_s$ && $Fr_s \neq s$ **then**
      $Fr_i \leftarrow i$ {claim to be the representative node}
    **end if**
  **end if**
**end procedure**

**procedure** processBeeAgents($b_s^{xv}$,$i$)
  **if** $b_s^{xv}$ already visited $i$ or its hop limit reached **then**
    kill $b_s^{xv}$ {avoid loops}
  **else if** $b_s^{xv}$ is inside $FZ_s$ **then**
    $qd \leftarrow qd + \frac{l_{ip}}{b_{ip}}$ and $pd \leftarrow pd + p_{ip}$
    update IFZ entries $q_{ps} \leftarrow qd$ and $p_{ps} \leftarrow pd$
    update $qd \leftarrow \sum_{k \in N(i)} (q_{ks} \times g_{ks})$
    update $pd \leftarrow \sum_{k \in N(i)} (p_{ks} \times g_{ks})$
  **else**
    $qd \leftarrow qd + \frac{l_{ip}}{b_{ip}}$ and $pd \leftarrow pd + p_{ip}$
    update IFR entries $q_{pz} \leftarrow qd$ and $p_{pz} \leftarrow pd$
    update $qd \leftarrow \sum_{k \in N(i)} (q_{kz} \times g_{kz})$
    update $pd \leftarrow \sum_{k \in N(i)} (p_{kz} \times g_{kz})$
  **end if**
  **if** $b_s^{jv}$ already reached $i$ {$\forall j \neq x$} **then**
    kill $b_s^{xv}$
  **else**
    use priority queues to forward $b_s^{xv}$ to all neighbors of $i$ except $p$
  **end if**
**end procedure**

| | |
|---|---|
| $i$ | Current node |
| $n$ | Successor node of $i$ |
| $p$ | Predecessor node of $i$ |
| $s$ | source node of a packet |
| $d$ | destination node of a packet |
| $b_s^v$ | Bee agent of ID v launched by $s$ |
| $b_s^{xv}$ | Replica x of $b_s^v$ |
| $h_s$ | Hop limit for bee agents of $s$ |
| $hops$ | Current number of hops |
| $D_{sd}$ | Data packet launched by $s$ towards $d$ |
| $Fr_c$ | Foraging region containing node c |
| $FZ_c$ | Foraging zone of node c |
| $z$ | Representative node of $Fr_s$ |
| $w$ | Representative node of $Fr_d$ |
| $qd$ | queuing delay estimate of a bee agent from $p$ to $s$ |
| $pd$ | propagation delay estimate of a bee agent from $p$ to $s$ |
| Short_Limit | Hop limit for short distance bee agents (7 hops) |
| Long_Limit | Hop limit for long distance bee agents |
| $t$ | current time |
| $t_{end}$ | Time to end simulation |
| $\Delta t$ | Generation interval for bee agents (1 sec) |
| $\Delta h$ | Generation interval for hello packets |
| $b_{ip}$ | Estimated bandwidth of link from $i$ to $p$ |
| $p_{ip}$ | Estimated propagation delay from $i$ to $p$ |
| $l_{ip}$ | Size of normal queue at $i$ for $p$ (bits) |
| $n_i$ | number of packets received at node $i$ |
| $N$ | number of neighbors of a node |
| % | Modulo operator |
| $=$ | Logical comparison |
| Packet_Limit | packet limit for launching bee agents |

Table 3.3: Symbols used in the *BeeHive* algorithm

**Algorithm 2** *Packet switching and neighbor maintenance algorithms*

**procedure** switchDataPackets($D_{sd}$,$i$)
  **if** $d$ is within $FZ_i$ **then**
    consult IFZ of node $i$ to find delays to node $d$
    calculate $g_{kd}$, $\forall k \in N(i)$
  **else**
    consult FRM of node $i$ to find node $w$
    consult IFR of node $i$ to find delays to node $w$
    calculate $g_{kw}$, $\forall k \in N(i)$
  **end if**
  probabilistically select a neighbor n ($n \neq p$) as per goodness
  enqueue data packet $D_{sd}$ in normal queue for neighbor $n$
**end procedure**

**procedure** manageNeighbors(i)
  **if** t % $\Delta h = 0$ **then**
    send a hello packet to all neighbors
    **if** time out before a response from neighbor {4th time} **then**
      neighbor is down
      update the routing table at $i$
      launch special bees to inform other nodes in $FZ_i$
    **end if**
  **end if**
**end procedure**

## 3.6 The performance evaluation framework for Nature inspired routing algorithms

We now introduce our performance evaluation framework that we used for an unbiased evaluation of the algorithms presented in Section 3.7. We used the guidelines suggested by Hinningbottom in [94] as well as our discussions with the Cisco network engineers in our system management group, for defining the relevant performance parameters of our framework. A prototype version of the performance evaluation framework was introduced in [216]. Its conceptual block diagram is illustrated in Figure 3.4. The framework consists of two input modules: the *topology generator* and

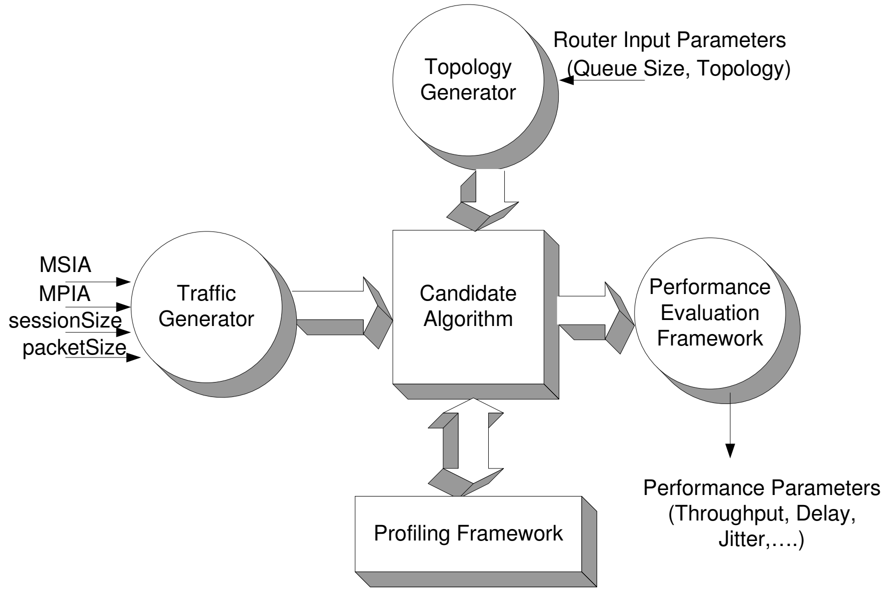



Figure 3.4: Performance evaluation framework

the *traffic generator*. The topology generator generates a topology of a given amount of nodes and links and assigns a buffer capacity to a router, and the traffic generator enables different traffic patterns through a variation of different input parameters. The traffic generator can generate session-oriented traffic in which all packets of a session have the same destination. This type of traffic tests the congestion control behavior of a routing algorithm. For injecting dynamically changing traffic patterns, we have defined two states: *uniform* and *weighted*. Each state lasts 10 seconds and then a state transition to another state occurs. In a *Uniform* state (U) a destination is selected from a uniform distribution while in a *Weighted* state (W), a destination selected in the previous *Uniform* state is favored over the other destinations.

$d_u(U) := \theta_d, (0 \leq \theta_d \leq 1)$

$d_w(W) := \theta_s(d_u) + (1-\theta_s)\theta_{d'}, (0 \leq \theta_{d'} \leq 1)$

$\theta_d$ represents the probability of selecting node $d$ as the destination of a packet in a session. In a (U) state, $\theta_d = 1/D$ where D is number of nodes in the network. During a (W) state, we are using $\theta_s = 0.4$, this means that we favor by 40% the destination from the previous *Uniform* state to be also the destination during the current *Weighted* state. The mechanism above ensures that

| | |
|---|---|
| MSIA | Mean of sessions inter-arrival times (sec) |
| MPIA | Mean of packets inter-arrival times (sec) |
| $sessionSize$ | The size of a session in bits |
| $D$ | The number of nodes in a network |
| $l_t$ | The number of bi-directional links in a network |
| $\beta_c$ | Buffer capacity (packets) of routers |
| $\delta_l$ | The size of a packet in bytes |

Table 3.4: Input parameter symbols used in the chapter

the traffic patterns could be dynamically generated in an arbitrary fashion in order to represent a decent subset of traffic patterns in real networks. The session-oriented traffic is shaped through the variables *sessionSize, MSIA and MPIA*. The size of a session is given by the sessionSize variable. MSIA is the mean of session inter-arrival times at a node and MPIA is the mean of packet inter-arrival times. The session inter-arrival and packet inter-arrival times are taken from negative exponential distributions with mean MSIA and MPIA, respectively. The input parameters and their symbolic representations are shown in Table 3.4.

In a session-less traffic, the destination of each packet is selected from a uniform distribution. This traffic pattern, under low load, simulates static network conditions. Generally researchers either use session-oriented or session-less traffic, though we believe that a *good routing algorithm should be able to do congestion control and be competitive under static network loads as well.*

This profiling framework has been developed to determine the processing complexity of the agents and the data packets for each routing algorithm. An ideal routing algorithm should be able to route data packets as quickly as possible, and the time it spends in processing the agents should be as small as possible. The framework is based on the *rdtsc* machine instruction supported by Pentium architectures [43] that provides cycle-level profiling accuracy. We decided to report the complexity of a an algorithm in cycles because this parameter is independent of the frequency of a processor. It is used as a standard parameter by the embedded systems community to report the complexity of an algorithm running on a hardware system.

Our framework keeps on logging the relevant parameters during the simulation, and it finally stores them into a data file. It measures a number of parameters which provide a comprehensive insight into the behavior of an algorithm, over a wide range of the operating environment. In this way we can evaluate each algorithm in an unbiased manner. The output parameters from the framework and their symbolic representations are shown in Table 3.5.

**Average Throughput**. Throughput is a measure of how much traffic is successfully received at the intended destination in a unit interval of time [94]. A routing protocol should try to maximize this value.

**Packet delay**. For all algorithms, we report the average packet delay and 90th percentile of the packet delays. A good algorithm should be able to deliver packets with minimum delay and with minimum standard deviation of delays. In the rest of the chapter we use the term packet delay for 90th percentile of packet delays, for the sake of brevity.

**Session delay**. Our network engineers suggested that in case of session-oriented traffic, the most important parameter is the time needed to complete a session. On the application layer at the destination node one only gets the packets after all packets of a session have been received in the correct order. The packet delay factors out this waiting time and hence favors multi-path routing algorithms which in general deliver packets in an out of order manner but with smaller delays.

**Session completion ratio**. The percentage of sessions that are able to complete without any support from transport layer protocols. For example, if only one packet in a session is dropped due to congestion or because the time-to-live (TTL) value becomes zero, we report the session as an incomplete one. We believe that this parameter indicates the drop pattern of packets in the face of congestion. Our results substantiate that it is more difficult to maximize this parameter than throughput.

**Packet delivery ratio**. This measure tells us how much of the data packets are successfully delivered at their destination. Under high loads a 1% improvement in packet delivery ratio at

| | |
|---|---|
| $T_{av}$ | Average throughput (Mbits/sec) |
| $P_d$ | Packet delivery ratio (%) |
| $P_{drop}$ | Packet drop ratio (%) |
| $P_{loop}$ | Percentage of packets that followed a cyclic path |
| $S_c$ | Session completion ratio (%) |
| $t_d$ | Average packet delay (msec) |
| $t_{90d}$ | 90th percentile of packet delays (msec) |
| $S_d$ | Average session delay (msec) |
| $S_{90d}$ | 90th percentile of session delays (msec) |
| $J_d$ | Average jitter value (msec) |
| $J_{90d}$ | 90th percentile of jitter times (msec) |
| $q_{av}$ | Average queuing delay (msec) |
| $A_a$ | Average agent processing cycles |
| $D_a$ | Average data processing cycles |
| $A_t$ | Total agent processing cycles (in billions ) per node |
| $D_t$ | Total agent processing cycles (in billions ) per node |
| $R_o$ | Control overhead |
| $S_o$ | Suboptimal overhead |
| $h_i^{sd}$ | hops packet i took to reach from node s to node d |
| $h_o^{sd}$ | minimum hops needed to reach from node s to node d |
| $h_{av}$ | Average hops count of the data packets |

Table 3.5: Output parameter symbols used in the chapter

times may mean about a few 100,000 more data packets delivered at their destination. Again, one can not observe this improvement via throughput values only.

**Packet loop ratio**. The percentage of data packets that followed a cyclic path. A cyclic path represents an error in a routing algorithm and should be reported but we do not have to kill such packets. The motivation for this approach is that a good stochastic routing algorithm must be able to quickly recover from such looping and if a packet happens to loop infinitely then it is dropped once its TTL value becomes zero.

**Jitter.** The jitter is defined as the difference in arrival times of two subsequent packets of the same session, sent from the same node, at their destination node. Let packet $p_i^{sd}$ and $p_{i+1}^{sd}$ be two subsequent packets sent from node $s$ to node $d$ and let $t_i$ and $t_{i+1}$ be their arrival times at destination node $d$. Now the jitter can be defined as $J_d = t_{i+1} - t_i$ if $t_{i+1} > t_i$ else $J_d = 0$. The jitter is an important parameter in quality of service (QoS) routing, especially for streaming multimedia applications. The subsequent packets in video streams should arrive with an acceptable jitter in order to avoid flickering in relaying the video.

**Average Agent Processing Cycles.** This parameter defines the processing complexity of an agent, say ant or bee. The results show a tendency that can be easily reproduced from one run to another within an acceptable variance. This parameter gives an insight into the complexity of computations/actions an agent performs at a node.

**Average Data Processing Cycles.** This parameter defines the processing complexity of switching a data packet to its next hop. It creates a considerable overhead for a routing algorithm, and should therefore be as small as possible.

**Total Agent Processing Cycles per node.** This parameter provides information about the total number of cycles (in billions) that a node spends in processing the agents. We will see the effect of accumulating small differences in average agent processing cycles over a longer period of time.

**Total Data Processing Cycles per node.** This parameter provides information about the total number of cycles (in billions) that a node spends in processing data packets. We will also see the effect of accumulating small differences in average data processing cycles over a longer period of time.

**Control overhead**. The ratio of the bandwidth occupied by the routing/control packets and the total available bandwidth in the network [52]. Generally, the authors report this parameter in order to show the control overhead of their routing algorithm.

**Suboptimal overhead**. This metric was introduced in [165], in the context of MANETs. We

believe that it is equally relevant in fixed networks. It is defined as *"The difference between the bandwidth consumed when transmitting data packets from all the sources to destinations and the bandwidth that would have been consumed should the data packets have followed the shortest hop count path"*. Formally we define the parameter as

$$S_o = \frac{\sum_{d=1}^{D} \sum_{s=1}^{D} \sum_{i=1}^{K} (h_i^{sd} - h_o^{sd}) \times L_i^{sd}}{B_t} \quad , s \neq d \tag{3.8}$$

where $D$ is total number of nodes in the network, $K$ is the total number of packets generated, $L_i^{sd}$ is the length of packet $i$ (in bits) from source $s$ to destination $d$, and $B_t$ is the total bandwidth of the network. We report this parameter because it implicitly includes the additional bandwidth consumed by data packets when they follow cyclic paths.

## 3.7 Routing algorithms used for comparison

The focus of our research is on adaptive (flexible) routing algorithms, but we will use *OSPF* in our comparative simulation for the sake of comprehensiveness.

### 3.7.1 AntNet

*AntNet* has been described in detail in Chapter 2. In rest of the chapter, we use the name *AntNet-CL* for the version of *AntNet* in which a Forward_Ant agent uses the same queues that data packets use while *AntNet-CO* corresponds to the version of *AntNet* in which a Forward_Ant agent utilizes priority queues and uses the estimation model described in [55] to estimate its trip time to the neighbor node. In the rest of the chapter, if not otherwise specified, we will refer to *AntNet-CO* as *AntNet* for the sake of the brevity.

### 3.7.2 DGA

*DGA* has been described in detail in Chapter 2. The poor performance of *DGA* under high traffic loads, as reported in [221], led us to investigate the problem in more detail. Our research revealed that launching half of the population of agents during the initialization phase resulted in approximately 50% control overhead which is not tolerable. Therefore, through rigorous testing we empirically reached a value of 12 to 16 ant agents, that should be launched during initialization phase, which reduced the control overhead to about 5 % without any significant performance degradation. Our experiments also suggest that in this way we have significantly improved the performance parameters of *DGA* as compared to those of the original algorithm.
Please note that in contrast to the above-mentioned algorithms, the *bee agents*, as discussed in Section 3.1, need not be equipped with a stack to perform their duties. Moreover, our agent model requires only forward moving agents, and they utilize an estimation model to calculate the trip time from their source to a given node. This model eliminates the need for global clock synchronization among routers, and it is expected that for very large networks routing information can be disseminated quickly with a rather small overhead as compared to *AntNet*. Our agent model does not require storing the average trip time, the variance of trip times, and the best trip time for each destination at a node in order to determine the goodness of a neighbor with respect to a particular destination. Last but not least, *BeeHive* works with significantly smaller routing tables as compared to *AntNet*.

### 3.7.3 OSPF

*OSPF* has been described in Chapter 2. In our implementation we simply take the fixed propagation costs and do not allow the costs to be changed by the network administrators. As a result, our implementation makes *OSPF* a single-path non-adaptive routing protocol. Such approach was also taken by the authors of *AntNet*.

### 3.7.4 Daemon

The daemon algorithm is an ideal algorithm in which all routers can have access to a single global instance of the network. They can update the costs of the links of the network based on their current queue lengths. Each router has an instant access to the queue length of all other routers and it applies the Dijkstra algorithm, for each routing decision, to find the shortest path between the current node to the destination node. The router updates the cost of the link from node $i$ to node $j$ ($C_{ij}$) after queuing the data packet $D_{sd}$ in its network buffer interface by using the formula $C_{ij} = \frac{L_j^{sd} + l_{ij}}{b_{ij}} + p_{ij}$, where $L_j^{sd}$ is the length of $D_{sd}$ (in bits). The other parameters are defined in Table 3.3. The algorithm can not be implemented in real routers because of the control overhead required to transmit changes in the queue lengths, and the processing overhead of running Dijkstra's algorithm for each routing decision. Nevertheless, the algorithm provides a bench mark for the comparison of the algorithms.

## 3.8 Simulation environment for BeeHive

In order to evaluate our algorithm *BeeHive* in comparison with *AntNet*, *DGA* and *OSPF* and *Daemon*, we implemented all of them in the OMNeT++ simulator [203]. For *OSPF* we implemented a link state routing that implements the deterministic Dijkstra Algorithm [57]. For *AntNet* and *DGA* we used the same parameters that were reported by the authors in [52] and [120], respectively. *BeeHive*, *OSPF* and *Daemon* algorithms were given 30 seconds to initialize the routing tables. In comparison *DGA* and *AntNet* were given 50 seconds to initialize the routing tables for the experiments reported in this chapter. Our simulation server was a Fujitsu Siemens machine with a Pentium 4 processor of 3 GHz having 1 Giga byte of main memory and 30 Giga byte of secondary memory. We tested the algorithms on the three network instances simpleNet, NTTNet and Node150. All reported values for an experiment are an average ($\mu$) of the values obtained from ten independent repetitions (each lasting 1000 seconds). The performance values of ten independent repetitions varied in the interval ($\mu \pm 5\%\mu$) for a confidence level of 95%.

### 3.8.1 simpleNet

simpleNet is a small network designed by the authors of [52] to study relevant aspects of a routing algorithm. We designed the experiments on simpleNet to study the effect of distributing network traffic loads on multiple paths, an important feature of *BeeHive* and *AntNet*. simpleNet is composed of 8 nodes and 9 bidirectional links each of 10 Mbits/sec bandwidth and 1 msec propagation delay. The topology is shown in Figure 3.5.

### 3.8.2 NTTNet

The next network instance that we used in our simulation framework is the Japanese Internet Backbone (NTTNet). It is a 57 nodes, 81 bidirectional links network. The Link bandwidth is 6 Mbits/sec, while propagation delays range from 1 to 5 msec. Moreover NTTNet it is a non-balanced oblong network with a low degree of connectivity. Hence it puts strong demands on the routing protocols because in a narrow shaped network, once a packet is forwarded in a wrong direction, it might not have a chance to be routed to the correct destination. NTTNet is shown in Figure 3.6. The advantages associated with the design options adopted in *BeeHive* are expected to become apparent from the topology of this size and onwards.

### 3.8.3 Node150

Our next network Node150 is a 150 nodes network with 200 bidirectional links. The link bandwidth is uniformly distributed between 6 and 10 Mbits/sec and the propagation delay is uniformly

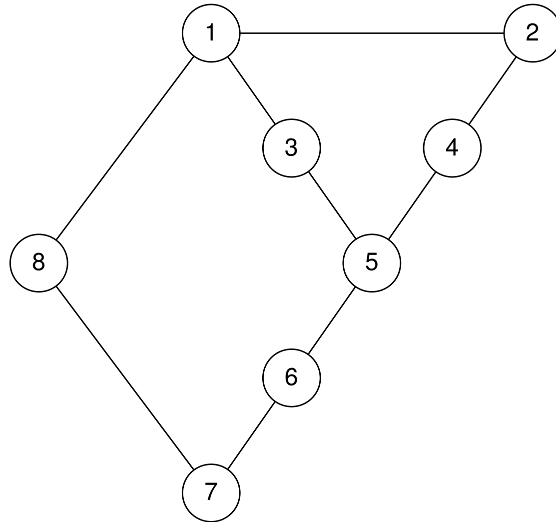



Figure 3.5: SimpleNet

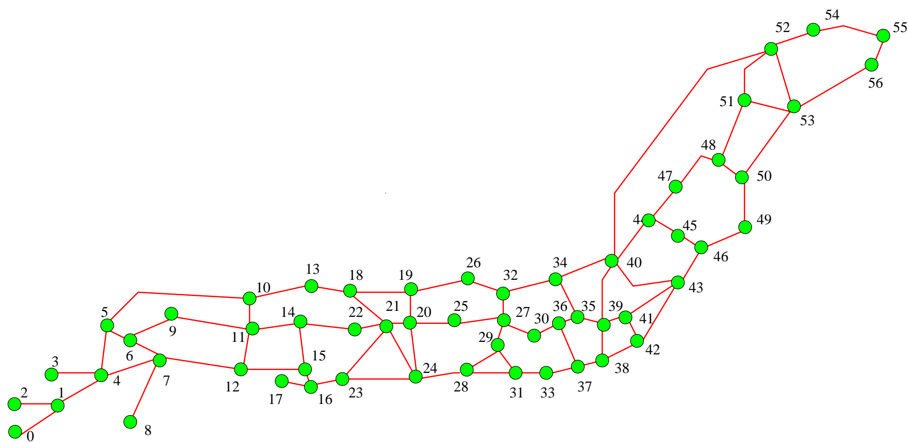



Figure 3.6: NTTNet

distributed between 1 and 5 msec. The topology was generated using the BRITE (Boston University Representative Internet Topology Generator) software developed by Medina at the University of Boston. Interested reader will find detailed information about the BRITE topology generator in [125, 126, 127]. The Node150 topology is shown in Figure 3.7.

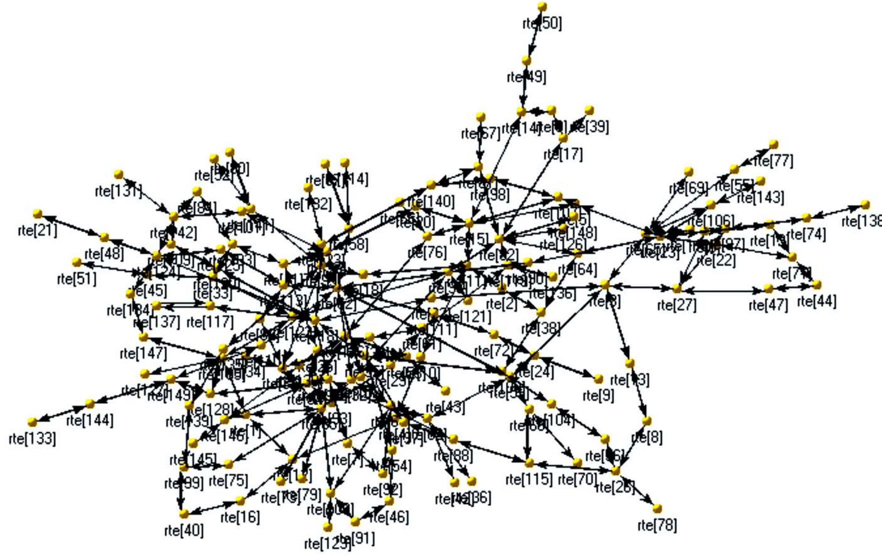



Figure 3.7: Node150: figure is captured from OMNeT++ plotter

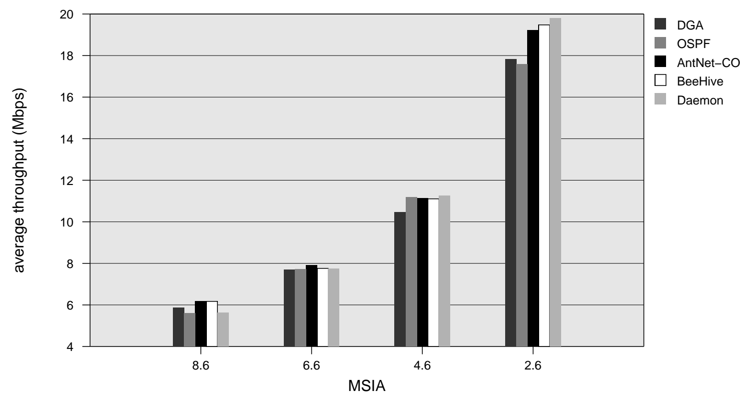
## 3.9 Discussion of the results from the experiments

### 3.9.1 Congestion control behavior

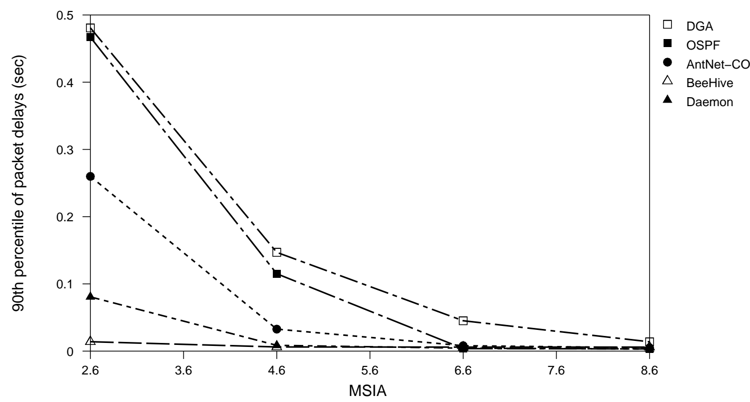
The purpose of the experiments, discussed in this subsection, was to study the congestion control behavior of the routing algorithms. The nearly saturated traffic load was created by gradually decreasing the value of MSIA from 8.6 sec to 1.6 sec (Please remember that MSIA and MPIA values are taken from negative exponential distributions). *The performance values of the algorithms in the following bar charts are represented from left to right for the algorithmic legends from top to bottom respectively.*

#### simpleNet

The performance values of all of the algorithms were approximately the same once we performed the experiments by changing MSIA from 8.6 sec to 1.6 sec, MPIA = 0.005 sec, sessionSize = 2130000 bits, $\delta_l = 512$ bytes and $\beta_c = 1000$ packets. Therefore, we designed a series of special experiments in which we enabled the traffic generators only at node 1 and node 7 (see Figure 3.5). We further ensured that all of the sessions originating at node 1 were having node 7 as the destination and vice-versa, in order to saturate the queues on selected paths. We increased the sessionSize to 26000000 bits and decreased MSIA from 8.6 sec to 2.6 sec while other parameters remained the same as discussed above. These traffic conditions did generate a challenging traffic pattern and showed the advantage of dynamic routing algorithms, *BeeHive*, *AntNet* and *Daemon*, over classical non-adaptive algorithms like *OSPF*. Please remember that *Daemon* has a complete knowledge about the topology and queue lengths of all routers in it and then it chooses the shortest path towards the destination while *BeeHive* and *AntNet* do a stochastic spread of the data packets based on the local information, collected by *bee* or *ant* agents respectively. Figure 3.8, Figure 3.9, Figure 3.10 and Figure 3.11 show the important parameters obtained from the experiments.

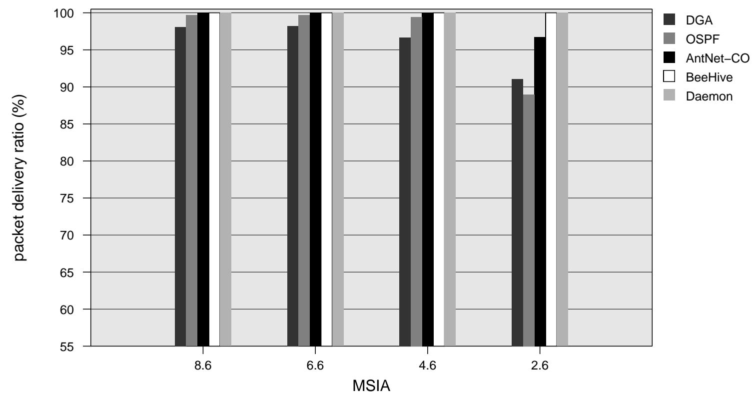



(a) Average throughput

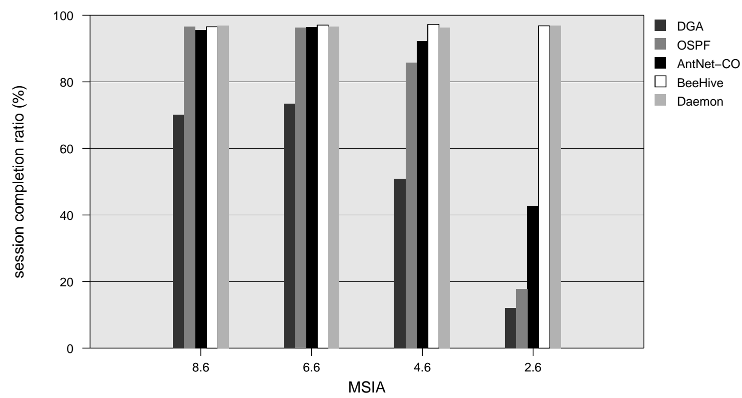



(b) 90th percentile of packet delays

Figure 3.8: Congestion control behavior in simpleNet (throughput and packet delay)

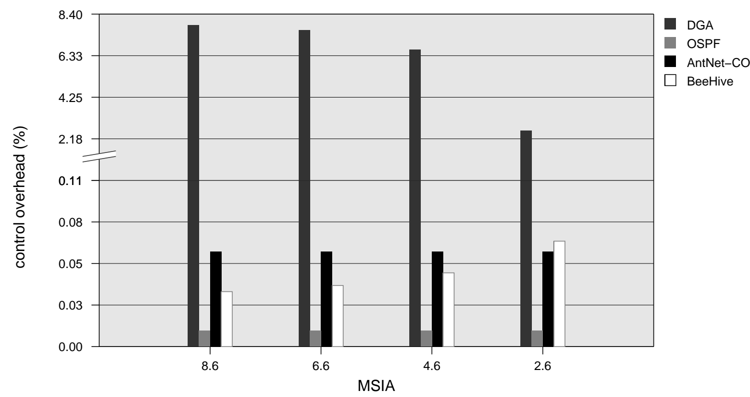



(a) Packet delivery ratio

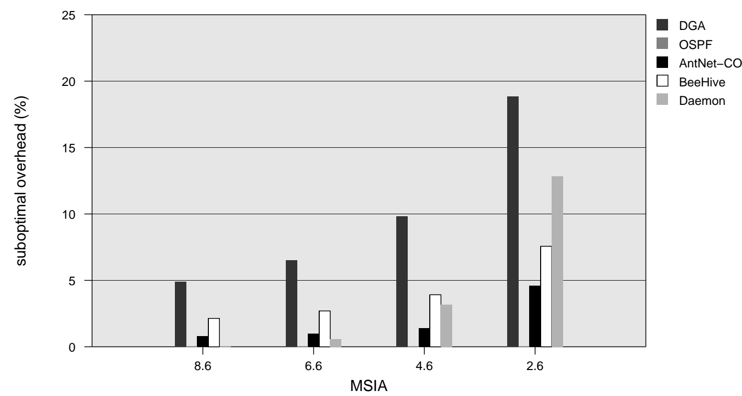



(b) session completion ratio

Figure 3.9: Congestion control behavior in simpleNet (packet delivery ratio and session completion ratio)

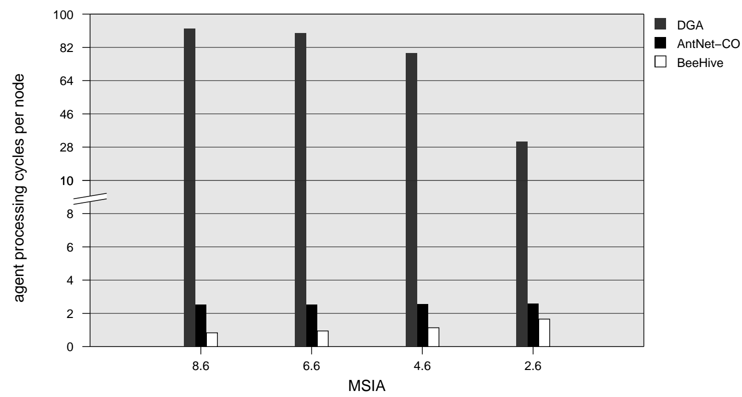



(a) Control overhead (%)

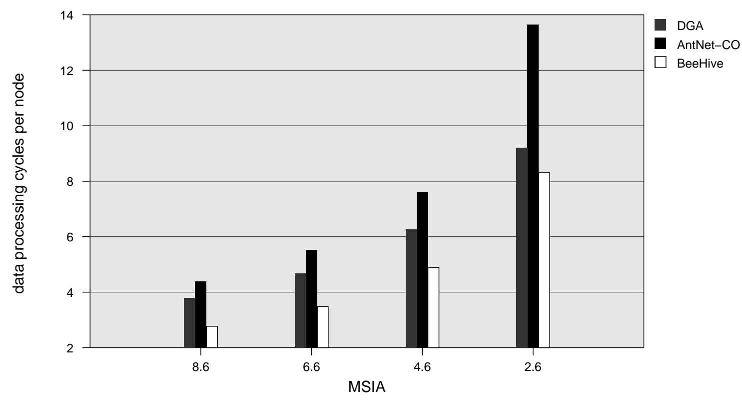



(b) Suboptimal overhead (%)

Figure 3.10: Congestion control behavior in simpleNet (control and suboptimal overhead)




(a) Total agent processing cycles per node (in billions)




(b) Total data processing cycles per node (in billions)

Figure 3.11: Congestion control behavior in simpleNet (agent and data processing complexity)

| MSIA | Algorithm | $t_d$ | $S_d$ | $S_{90d}$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $q_{av}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8.6 | DGA | 5.8 | 31736 | 32255 | 3.13 | 2 | 6 | 0.15 | 3.49 | 27787 | 4444 |
| | OSPF | 3 | 31732 | 32243 | 0 | 2 | 5.1 | 1 | 2 | - | - |
| | AntNet-CL | 3 | 31738 | 32262 | 0.01 | 2 | 5.2 | 0 | 2.24 | 69925 | 7244 |
| | AntNet-CO | 3 | 31719 | 32219 | 0.01 | 2 | 5.1 | 0 | 2.23 | 71791 | 7232 |
| | BeeHive | 3.75 | 31741 | 32250 | 0 | 2.7 | 6.2 | 0.03 | 2.61 | 38141 | 4077 |
| | Daemon | 2.96 | 31732 | 32242 | 0 | 2 | 5.1 | 0.07 | 2 | - | - |
| 6.6 | DGA | 9.43 | 31737 | 32247 | 2.72 | 1.6 | 7.2 | 0.63 | 3.51 | 27960 | 4186 |
| | OSPF | 3 | 31735 | 32242 | 0 | 1.2 | 3.9 | 0.5 | 2 | - | - |
| | AntNet-CL | 3.4 | 31736 | 32241 | 0.01 | 1.3 | 3.9 | 0.1 | 2.23 | 70183 | 7141 |
| | AntNet-CO | 3.4 | 31737 | 32244 | 0.01 | 1.4 | 3.9 | 0.2 | 2.22 | 71779 | 7112 |
| | BeeHive | 3.78 | 31724 | 32235 | 0 | 2 | 5.2 | 0.03 | 2.62 | 38898 | 4064 |
| | Daemon | 3.23 | 31734 | 32241 | 0 | 1.2 | 3.9 | 0.11 | 2.12 | - | - |
| 4.6 | DGA | 31.62 | 31753 | 32267 | 4.81 | 1.7 | 14.4 | 3.79 | 3.68 | 28153 | 3854 |
| | OSPF | 21.2 | 31742 | 32249 | 0 | 1 | 2.2 | 9.5 | 2 | - | - |
| | AntNet-CL | 7.7 | 31733 | 32249 | 0.02 | 1 | 4.5 | 1.8 | 2.25 | 70390 | 6952 |
| | AntNet-CO | 6.6 | 31744 | 32261 | 0.01 | 1 | 4.6 | 1.6 | 2.22 | 72353 | 6950 |
| | BeeHive | 3.89 | 31738 | 32256 | 0 | 2 | 4 | 0.07 | 2.62 | 38657 | 3979 |
| | Daemon | 4.6 | 31741 | 32246 | 0 | 1 | 2.3 | 0.43 | 2.5 | - | - |
| 2.6 | DGA | 187.74 | 31739 | 32297 | 2.08 | 1.7 | 24.6 | 38.19 | 3.89 | 28397 | 3284 |
| | OSPF | 224.5 | 31768 | 32300 | 0 | 0.8 | 1 | 110.9 | 2 | - | - |
| | AntNet-CL | 79.4 | 31762 | 32302 | 0.14 | 2.2 | 25.6 | 31.1 | 2.45 | 71802 | 6727 |
| | AntNet-CO | 87.1 | 31750 | 32268 | 0.13 | 2.1 | 25.2 | 34.4 | 2.43 | 73728 | 6715 |
| | BeeHive | 5.51 | 31728 | 32242 | 0 | 1 | 4.6 | 0.64 | 2.69 | 39250 | 3789 |
| | Daemon | 32.52 | 31762 | 32279 | 0 | 0.8 | 1 | 8.87 | 3.15 | - | - |

Table 3.6: Performance parameters for congestion control behavior in simpleNet

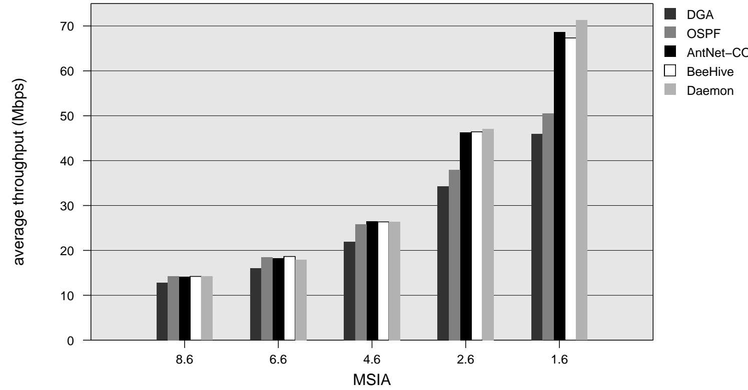
Please note that as we move close to saturated traffic loads, the packet delivery ratio of *OSPF* and *DGA* starts trailing significantly behind as compared to that of *BeeHive*, *AntNet* and *Daemon* algorithms. However, Figure 3.9(b) shows that the session completion ratio of *OSPF*, *DGA* and *AntNet* is significantly smaller than that of *BeeHive* and *Daemon* algorithms. This confirms our expectation (see Section 3.6): *during congestion, a small difference in packet delivery ratio results in a significantly larger difference in session completion ratio.* Please note that *BeeHive* and *Daemon* are able to maintain higher throughput, smaller packet delay and higher session completion ratio than the other algorithms, under all the simulated scenarios.

Figure 3.10 shows that the control overhead and suboptimal overhead of *BeeHive* and *AntNet* are approximately the same. The suboptimal overhead of *Daemon* is the largest because it tries to distribute the packets on all available paths based on its global information about the network, as a result, the average hop count increases to about 3.0 hops. The additional fractional increase in $h_{av}$ at extreme saturating loads translates to a significant increase in suboptimal overhead. The control overhead of *DGA* decreases with an increase in traffic load because it employs a genetic algorithm. Please remember that in *DGA*, ant agents use the same buffers as data packets, and next generations of agents are launched once the node receives four agents from the previous generation. Consequently, more agents will be traveling on the network if their trip time is smaller and this happens in a small topology or under low traffic loads, or both. This explains a sharp increase in the control overhead with an increase in the network traffic.

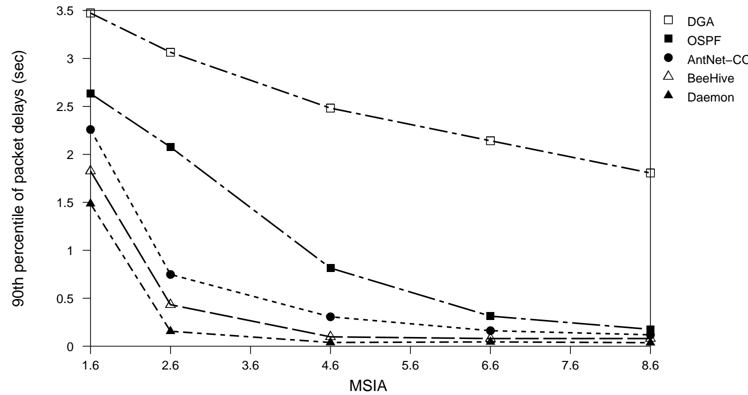
Figure 3.11 shows the parameters that provide an insight into the processing complexity of the agents and the efficiency with which a data packet can be switched. The number of cycles that a node spends in processing *bee agents* is smaller than *ant agents* in *AntNet*, and the packet switching complexity of *BeeHive* is significantly smaller than *AntNet*. These are the benefits of employing the simple *bee agents*, and of maintaining small routing tables (see Figure 3.1). We have collected all other important parameters in Table 3.6.

**NTTNet**

We continued our study of congestion control behavior on NTTNet, a relatively complex network topology. We enabled the traffic generators on all of the nodes with the following parameters: MPIA = 0.005 sec, sessionSize = 2130000 bits, $\delta_l$ = 512 bytes and $\beta_c$ = 1000 packets. We decreased the values of MSIA from 8.6 to 1.6 sec. Figure 3.12 and Figure 3.13 show the important




(a) Average throughput

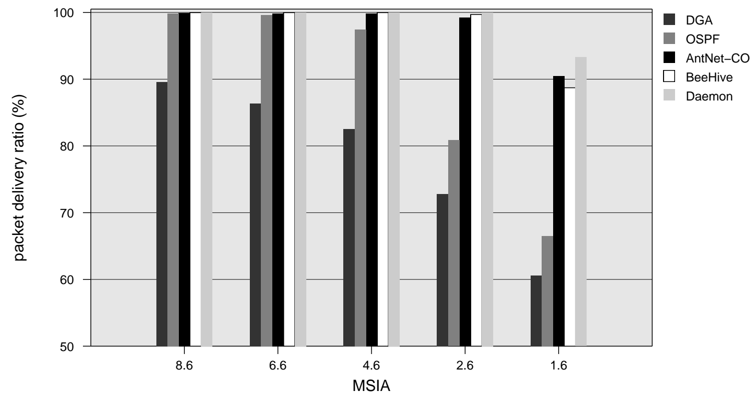



(b) 90th percentile of packet delays

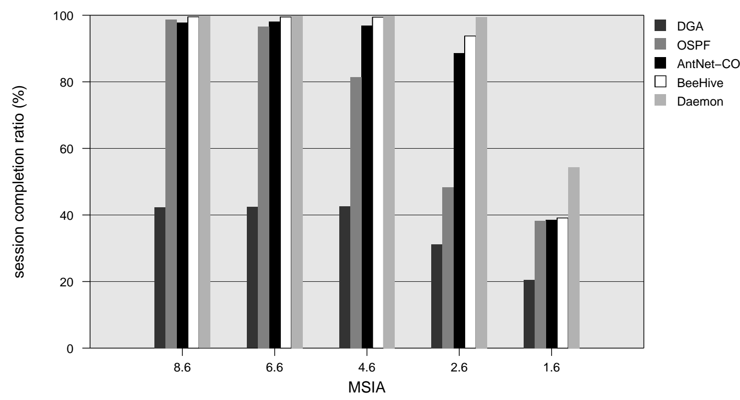
Figure 3.12: Congestion control behavior in NTTNet (throughput and packet delay)

performance parameters. One can easily conclude from the figures that *BeeHive*, *AntNet* and *Daemon* algorithms scale nicely with an increase in the traffic load. The throughput, packet delay and packet delivery ratio of *BeeHive* and *AntNet* are close to *Daemon*. However, session completion ratio, as expected, of all algorithms degrades significantly at MSIA = 1.6 sec. Please note that *OSPF* and *DGA* are unable to cope with the congestion but the performance of *DGA* is worst among all algorithms.

From Figure 3.14 it is clear that *BeeHive* has significantly smaller control and suboptimal overhead as compared to *AntNet* under all of the conditions. *OSPF* has the smallest control overhead and suboptimal overhead but then one should try to complete the picture with other parameters such as throughput, packet delay and packet delivery ratio shown in Figure 3.12 and Figure 3.13. An important conclusion from Figure 3.14 is that the suboptimal overhead is an order of magnitude higher than the control overhead, and hence should be taken into account during the design of an adaptive routing algorithm. This parameter can be optimized if an algorithm brings more packets

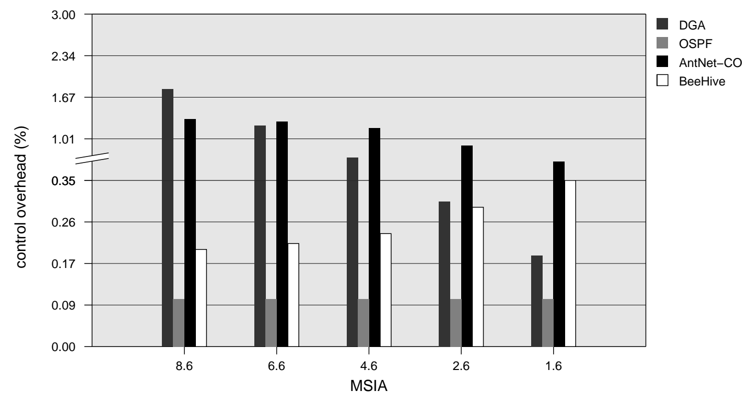



(a) Packet delivery ratio

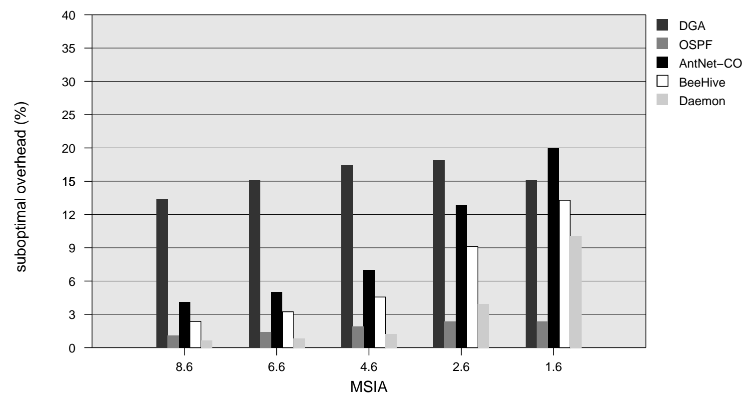



(b) session completion ratio

Figure 3.13: Congestion control behavior in NTTNet (packet delivery ratio and session completion ratio)

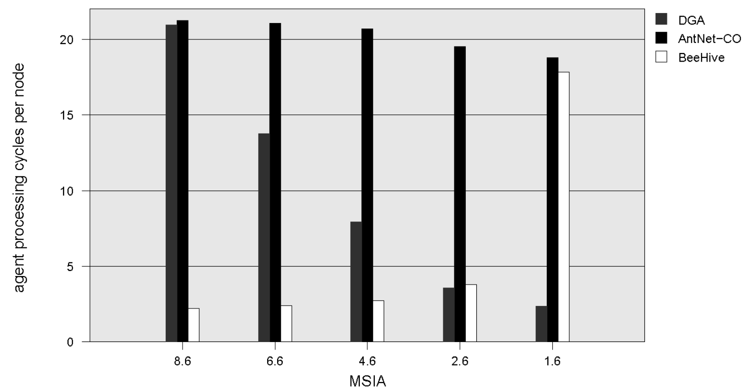



(a) Control overhead (%)

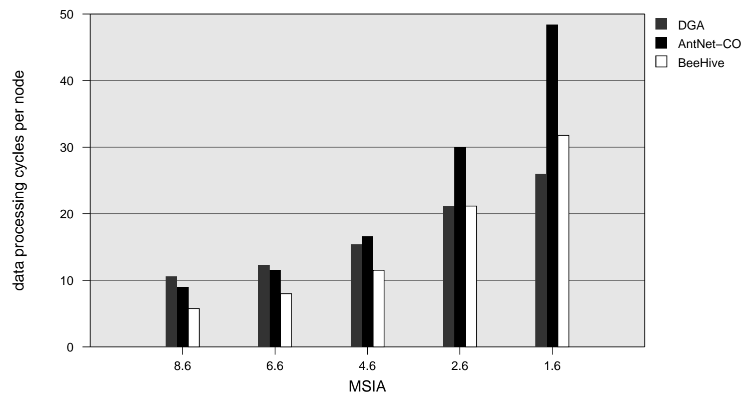



(b) Suboptimal overhead (%)

Figure 3.14: Congestion control behavior in NTTNet (control and suboptimal overhead)




(a) Total agent processing cycles per node (in billions)




(b) Total data processing cycles per node (in billions)

Figure 3.15: Congestion control behavior in NTTNet (agent and data processing complexity)

| MSIA | Algorithm | $T_d$ | $S_d$ | $S_{90d}$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $q_{av}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8.6 | DGA | 469.31 | 2803 | 3531 | 30.44 | 102.2 | 699.8 | 29.97 | 15.97 | 52448 | 8341 |
| | OSPF | 38.8 | 2634 | 2813 | 0 | 5 | 11 | 9.2 | 6.9 | - | - |
| | AntNet-CL | 32.6 | 2630 | 2791 | 4.38 | 6.9 | 53.3 | 1 | 9 | 75622 | 14655 |
| | AntNet-CO | 32 | 2630 | 2792 | 4.39 | 6.9 | 42.8 | 0.9 | 8.97 | 79515 | 14679 |
| | BeeHive | 25.28 | 2622 | 2773 | 2.45 | 6.7 | 20.9 | 0.35 | 7.74 | 25109 | 10825 |
| | Daemon | 20.74 | 2617 | 2765 | 0 | 4.9 | 10.9 | 0.1 | 6.62 | - | - |
| 6.6 | DGA | 587.24 | 2825 | 3578 | 27.61 | 114.1 | 784.8 | 44.42 | 14.86 | 50968 | 8176 |
| | OSPF | 75.1 | 2664 | 2907 | 0 | 5 | 11 | 15.8 | 6.88 | - | - |
| | AntNet-CL | 38.6 | 2637 | 2818 | 3.61 | 7 | 54.1 | 1.8 | 8.92 | 77933 | 14745 |
| | AntNet-CO | 38.3 | 2636 | 2811 | 3.65 | 6.9 | 49.8 | 1.5 | 8.82 | 81876 | 14819 |
| | BeeHive | 26.03 | 2622 | 2773 | 2.56 | 6.7 | 20.4 | 0.4 | 7.86 | 25889 | 11288 |
| | Daemon | 22.02 | 2620 | 2769 | 0 | 4.9 | 10.9 | 0.26 | 6.53 | - | - |
| 4.6 | DGA | 738.29 | 2850 | 3582 | 23.17 | 113.9 | 801.8 | 65.91 | 13.26 | 50606 | 8121 |
| | OSPF | 254.3 | 2736 | 3108 | 0 | 4 | 10 | 49.7 | 6.8 | - | - |
| | AntNet-CL | 69.4 | 2669 | 2917 | 2.64 | 9.2 | 79.3 | 5.1 | 8.76 | 84837 | 14977 |
| | AntNet-CO | 72.5 | 2673 | 2933 | 2.88 | 9.3 | 80.9 | 5.6 | 8.7 | 88373 | 14974 |
| | BeeHive | 30.1 | 2629 | 2788 | 2.5 | 7 | 28.6 | 0.95 | 7.81 | 26624 | 11558 |
| | Daemon | 21.51 | 2620 | 2768 | 0 | 4 | 10 | 0.2 | 6.65 | - | - |
| 2.6 | DGA | 1078.9 | 2883 | 3559 | 15.03 | 99.3 | 747.1 | 121.19 | 10.45 | 53028 | 8183 |
| | OSPF | 751.3 | 2677 | 2962 | 0 | 4 | 9.9 | 149.8 | 6.21 | - | - |
| | AntNet-CL | 235.4 | 2824 | 3303 | 2.65 | 26.6 | 182.9 | 23.9 | 8.82 | 104061 | 15220 |
| | AntNet-CO | 235.7 | 2825 | 3310 | 2.76 | 26.1 | 182.9 | 23.9 | 8.83 | 107924 | 15249 |
| | BeeHive | 125.26 | 2755 | 3143 | 3.79 | 27.9 | 162.5 | 12.67 | 8.06 | 29733 | 11702 |
| | Daemon | 45.83 | 2647 | 2836 | 0 | 5 | 13 | 3.56 | 6.99 | - | - |
| 1.6 | DGA | 1363.84 | 2861 | 3382 | 8.32 | 69.6 | 610 | 190.59 | 8.1 | 55964 | 8317 |
| | OSPF | 896.6 | 2695 | 2989 | 0 | 3 | 8 | 212.9 | 5.53 | - | - |
| | AntNet-CL | 987.1 | 2946 | 3550 | 3.03 | 82 | 448.3 | 113.9 | 8.59 | 117603 | 15876 |
| | AntNet-CO | 1001.8 | 2985 | 3637 | 3.38 | 64.4 | 356.8 | 112.3 | 8.77 | 138650 | 15824 |
| | BeeHive | 721.16 | 2854 | 3391 | 4.48 | 116.3 | 551.2 | 92.19 | 7.79 | 108788 | 11742 |
| | Daemon | 666.2 | 2880 | 3372 | 0 | 10 | 42 | 87.64 | 7.36 | - | - |

Table 3.7: Performance parameters for congestion control behavior experiments for NTTNet

to a destination, with less hops and with a small $p_{loop}$ value. This parameter, to our knowledge, has received little attention in the Nature inspired routing community.

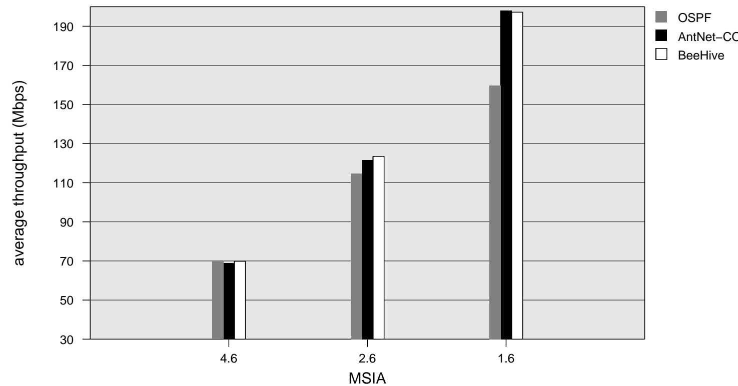
Figure 3.15 shows the agent processing and data packet processing complexities. The simple design of *bee agents* which is a consequence of no stack processing, no complex mathematical formula evaluation and only forward moving agents, now starts showing its benefits because one can see an order of magnitude difference between the total time that a node spends in processing the agents. The processing of the agents in *AntNet* takes approximately 20 billion cycles as compared to 5 billion cycles taken by processing of *bee agents*. One can easily note the sharp increase in the processing complexity of *bee agents* to 18 billion cycles at MSIA = 1.6 sec. This is a consequence of a significant increase in the average processing complexity of *bee agents* from 30000 cycles to 108788 cycles (see Table 3.7). The increase in the average cycle count appears to be counterintuitive as the actions that agents take remain the same as in the previous cases. We investigated the problem and it appeared that under saturated conditions the event handling mechanism of OMNeT++ is time consuming especially if a packet needs to be flooded. All of the packets except *bee agents* follow point-to-point traversal of the network, therefore, the average processing complexity for *ant agents* and *data packets* remain approximately the same. Our conclusion is that under extremely saturated conditions specially for MSIA = 2.0 sec and below the results of the processing complexity might not be adequate because of the event-handling mechanism of OMNeT++. Apart from MSIA = 1.6 sec, the average processing complexity of *bee agents* is always between 20000 to 35000 cycles, which is reasonably acceptable. The other important parameters are collected in Table 3.7.

Please note that *DGA* has the same behavior as observed in the simpleNet topology: the control overhead significantly decreases as the network traffic load is increased by decreasing the MSIA. The performance of *DGA* is again the worst. However, our improvements in *DGA* significantly improved its performance as compared to the original *DGA* (see Section 3.7). The performance

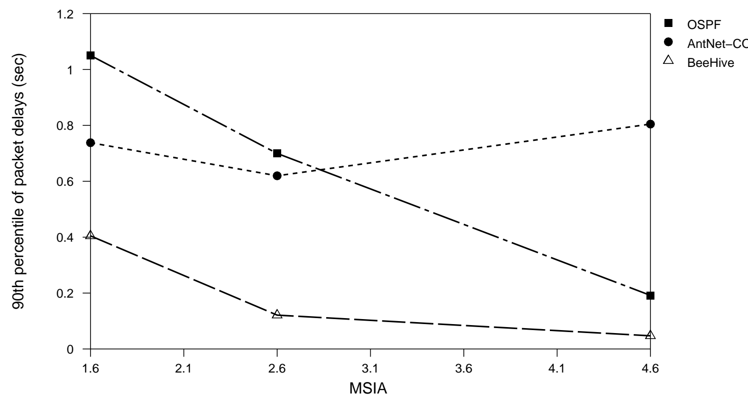
of the original *DGA* is reported in [221].

### Node150

The time required to simulate the algorithms on a Node150 network increased exponentially. Therefore, we decided to select the algorithms performing best on the smaller topologies for simulation on the Node150 network. *AntNet-CO* is designed for quick spreading of the routing information, therefore, we did not simulate its counterpart *AntNet-CL* on a Node150 network. The poor performance of *DGA* on small topologies made it an obviously weaker candidate which can be easily dropped. The time needed to simulate *Daemon* on NTTNet was significantly greater than those of the other algorithms, therefore, we did only one set of the experiments for *Daemon*. We also think that it makes sense to drop *Daemon* from our short list as the algorithm is more or less used as a bench-mark, and it is impossible to implement it on any real network because of its communication/processing complexity. We did not drop *OSPF* because it is widely used routing algorithm in the Internet and we want to always take it as a reference point (this is a shortcoming in the *DGA* presentations). We believe that *AntNet-CO* and *BeeHive* are two clear winners from the experiments on smaller topologies and hence it makes perfect sense to compare their performance with one another. Figure 3.16, Figure 3.17, Figure 3.18 and Figure 3.19



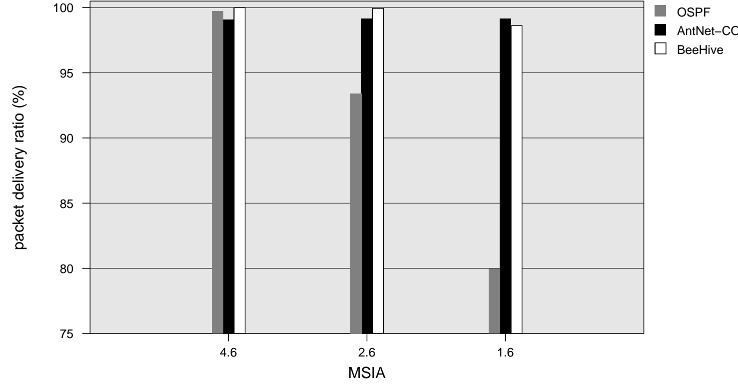

(a) Average throughput



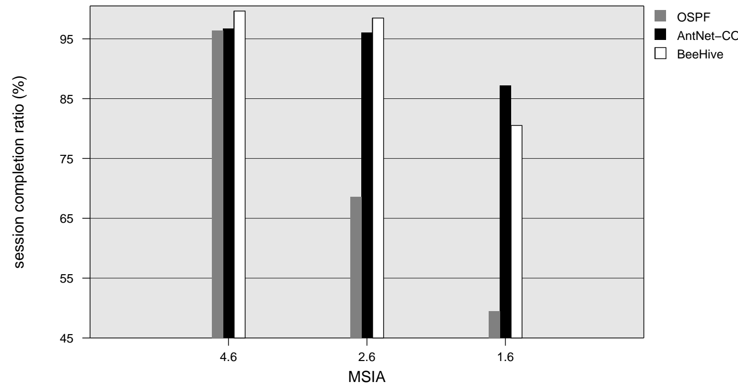

(b) 90th percentile of packet delays

Figure 3.16: Congestion control behavior in Node150 (throughput and packet delay)

show the same behavior of *AntNet* and *BeeHive* as in the previous congestion control experiments. Both algorithms are able to deliver more packets and complete more sessions with an increase in




(a) Packet delivery ratio

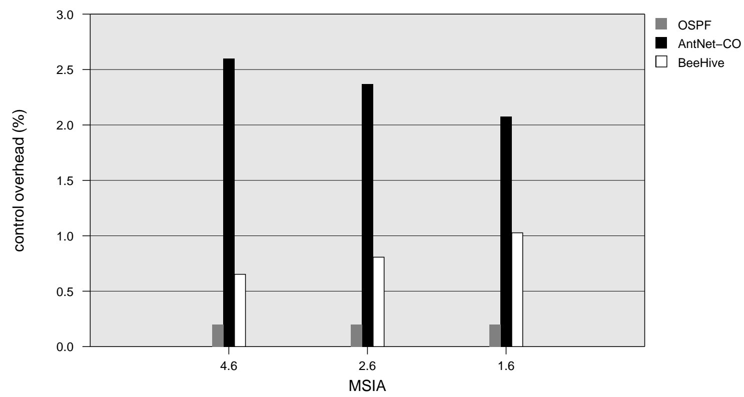



(b) session completion ratio

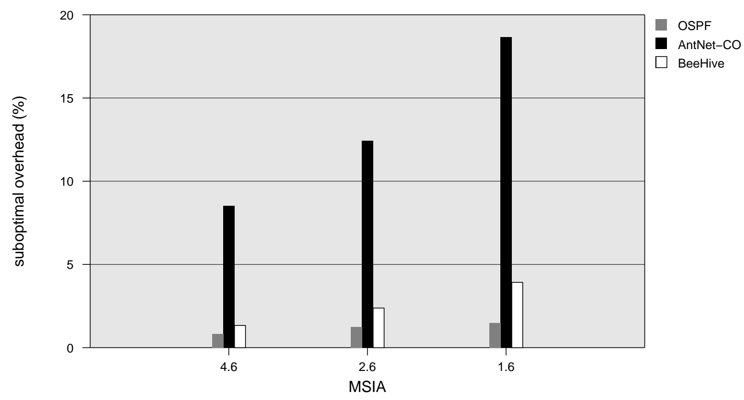
Figure 3.17: Congestion control behavior in Node150 (packet delivery ratio and session completion ratio)

| MSIA | Algorithm | $T_d$ | $S_d$ | $S_{90d}$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $q_{av}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.6 | OSPF | 48.8 | 2638 | 2820 | 0 | 4 | 10 | 16.2 | 5.4 | - | - |
| | AntNet-CO | 111.6 | 2716 | 3395 | 2.46 | 13.4 | 188 | 12.6 | 8.86 | 110235 | 22808 |
| | BeeHive | 22.12 | 2621 | 2769 | 0.59 | 6 | 15 | 0.3 | 5.63 | 34090 | 13648 |
| 2.6 | OSPF | 235.6 | 2663 | 2888 | 0 | 4 | 9.2 | 70.6 | 5.34 | - | - |
| | AntNet-CO | 109.2 | 2710 | 3134 | 1.73 | 14.6 | 146.6 | 11.6 | 8.12 | 112666 | 23120 |
| | BeeHive | 36.77 | 2642 | 2817 | 0.6 | 7 | 35.8 | 2.88 | 5.62 | 34399 | 13399 |
| 1.6 | OSPF | 384 | 2668 | 2891 | 0 | 3 | 8 | 107.6 | 5.18 | - | - |
| | AntNet-CO | 223.2 | 2848 | 3324 | 1.66 | 30 | 221.4 | 24.8 | 7.9 | 119439 | 23894 |
| | BeeHive | 129.38 | 2732 | 3016 | 0.62 | 18.2 | 113.6 | 19.38 | 5.64 | 47451 | 13212 |

Table 3.8: Performance parameters for congestion control behavior in Node150

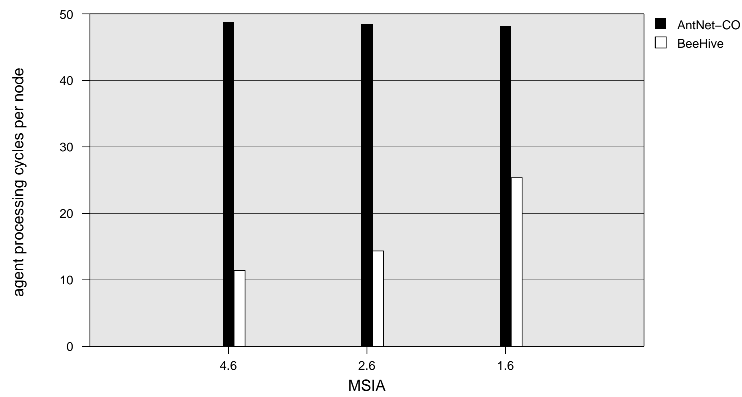



(a) Control overhead (%)

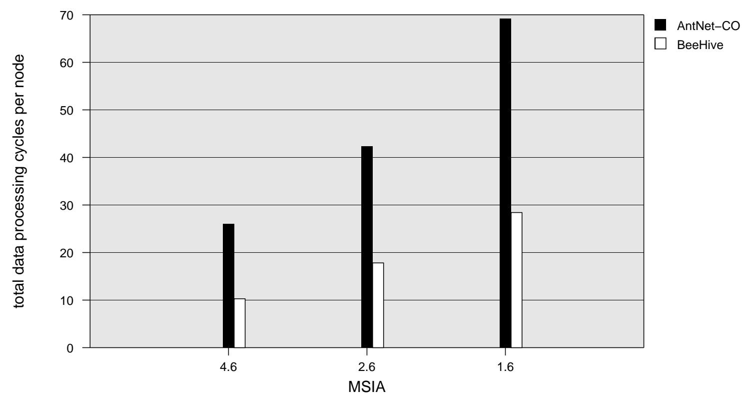



(b) Suboptimal overhead (%)

Figure 3.18: Congestion control behavior in Node150 (control and suboptimal overhead)




(a) Total agent processing cycles per node (in billions)




(b) Total data processing cycles per node (in billions)

Figure 3.19: Congestion control behavior in Node150 (agent and data processing complexity)

the network traffic load, as compared with *OSPF*. Please note that both packet delay and session delay of *BeeHive* are the smallest among the three algorithms. As expected, *OSPF* is not able to scale to increasing network traffic.

The benefit of collecting routing information in small *foraging zones* around a node in *BeeHive* is becoming more apparent as one looks at Figure 3.18(a). The control overhead of *BeeHive* is significantly smaller than *AntNet* and the feature of routing packets in less hops is manifesting its benefits in Figure 3.18(b). This significant difference in suboptimal overhead is due to the fact that a smaller number of data packets in *BeeHive* follow cyclic paths (see Table 3.8), and that *BeeHive* delivers data packets at their destination in less hops. One can easily conclude that the suboptimal overhead of *BeeHive* is now approaching to that of *OSPF*.
Figure 3.19 shows the processing complexity for agents and data packets. The simple behavior of *bee agents*, as discussed previously, is now showing significant benefits. The total time that a node spends in processing *bee agents* is approximately 1/5th of its time processing *ant agents*. The reasons for a significantly smaller suboptimal overhead of *BeeHive*, discussed in the previous paragraph, are also valid for the smaller packet switching complexity of *BeeHive* as compared to *AntNet*.

Please note that Node150 has links of 6-10 Mbits/sec bandwidth. As a result, the saturation of queue buffers at MSIA = 1.6 sec is not as visible as it was in the NTTNet experiments. Therefore, once we look at the average processing complexity of *bee agents*, it is around 47,000 cycles. This further strengthened our previous findings that an exponential increase in the average bee agent processing complexity in NTTNet at MSIA = 1.6 sec stems from specifics of the OMNeT++ simulator, and not from the *BeeHive* algorithm. *BeeHive* has clearly manifested its advantages on a Node150 network over *AntNet* and we believe that the benefits will be even more apparent on bigger topologies. We have collected other important performance parameters from these experiments in Table 3.8.

### 3.9.2 Queue management behavior

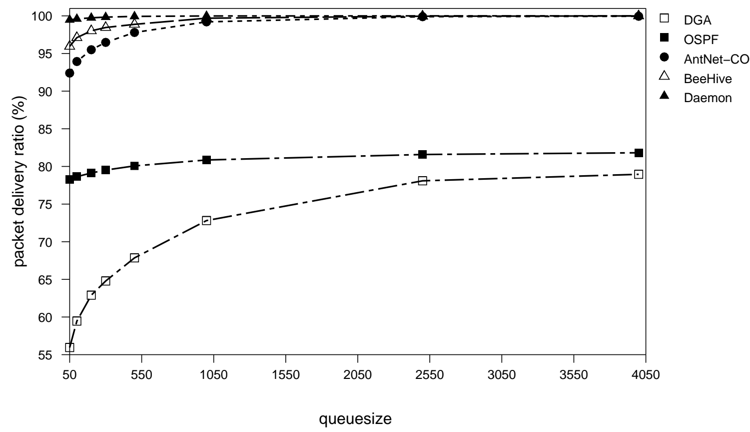
The purpose of these sets of experiments was to study the queue control behavior of the routing algorithms. We believe that it is important to know how the algorithms scale to different sizes of queue buffers. This is important for two reasons: one, to get an idea about an optimal queue buffer size for achieving the best performance, and two, to investigate the benefits that multi-path routing algorithms can bring on networks which have small devices like PDAs with limited main memory. Moreover, achieving better performance with small buffer capacities is a desirable property of any routing algorithm. During these experiments we kept MSIA = 2.6 sec, MPIA = 0.005 sec, sessionSize = 2130000 bits $\delta_l$ = 512 bytes and varied the buffer capacity $\beta_c$ from 50 packets to 4000 packets.

#### simpleNet

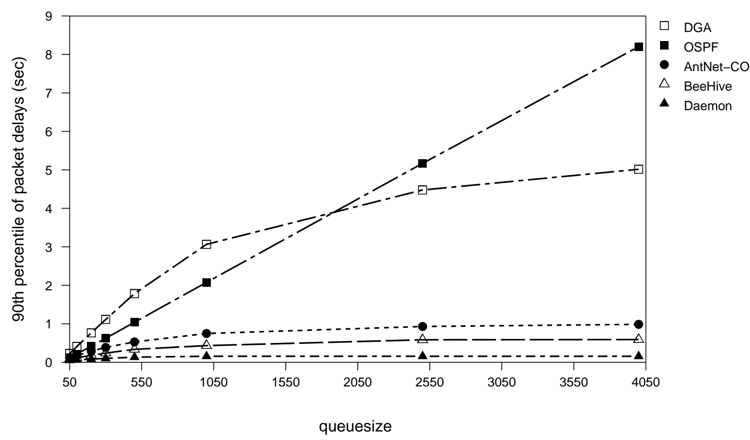
One can not see a significant difference between performance values in simpleNet topology. Therefore, we skip the results for the sake of brevity.

#### NTTNet

The behavior of the algorithms during these experiments is summarized in Figure 3.20 and Figure 3.21. One can see from Figure 3.20(a) that *BeeHive* is able to deliver about 2% more packets than *AntNet* for smaller buffer capacities, however, *AntNet* is able to catch up with *BeeHive* at a queue size of 1000 packets. Please note that 2% more packet deliery ratio as compared to *AntNet* at $\beta_c = 50$ results in about 15% more sessions completed (see Figure 3.21(a)), which is a significant improvement and shows the superiority of *BeeHive* over *AntNet* for low buffer capacities. Figure 3.20(b) shows the real shortcoming of classical non-adaptive algorithms like *OSPF* where an increase in buffer capacity does not result in any significant increase in packet delivery ratio or session completion ratio. This shows that *OSPF* is unable to manage higher traffic loads because of lack of its queue management. The performance of *DGA* improves with an increase in buffer capacity but is far inferior to *BeeHive* or *AntNet*. *BeeHive* has a better scalability as compared to all other algorithms except *Daemon*. On the other hand the *Daemon* algorithm has the best performance among all the algorithms. Another important observation from Table 3.9 is that the jitter value significantly increases for *BeeHive* and *AntNet* with an increase in buffer capacity. The reason is obviously that both algorithms stochastically spread data packets on multiple paths. As a result, subsequent packets might follow different paths. The difference in arrival time at destination increases with an increase in buffer capacity and this connects the increasing jitter behavior with an increase in buffer capacity. The average agent processing complexity ($A_a$) of *bee agents* sharply increases at smaller buffer capacities, and the reason is similar to some OMNeT++ related behavior discussed in the previous Section 3.9.1.
We have collected other important performance parameters in Table 3.9.

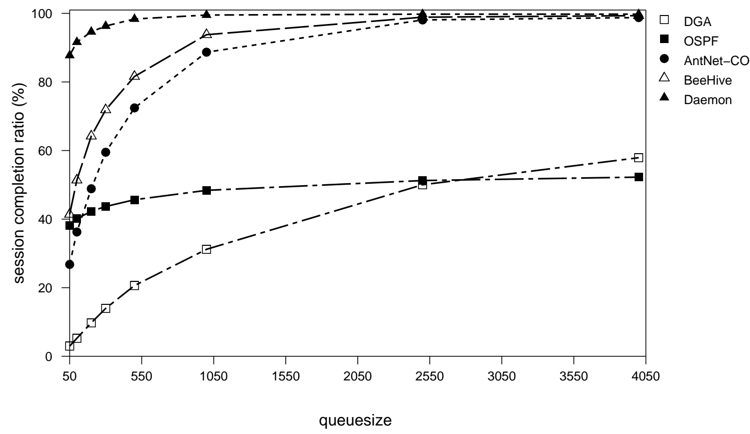



(a) Packet delivery ratio (%)

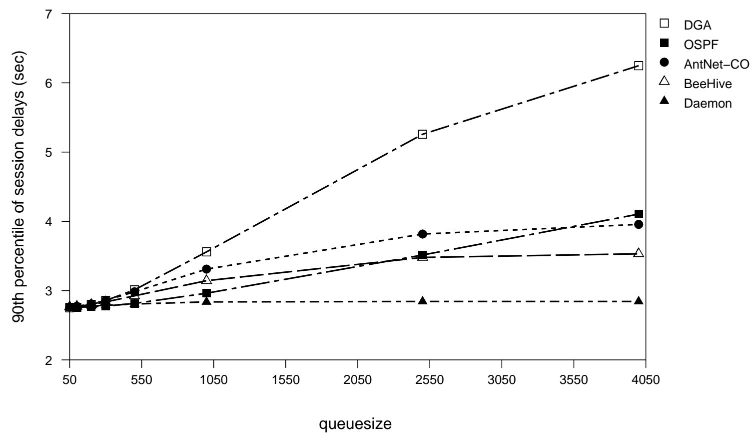



(b) 90th percentile of packet delays (sec)

Figure 3.20: Queue management/control behavior of algorithms (packet delivery ratio and packet delay)




(a) Session completion ratio (%)




(b) 90th percentile of session delays (sec)

Figure 3.21: Queue management/control behavior of algorithms (session completion ratio and session delay)

| $\beta_C$ | Algorithm | $T_{av}$ | $t_d$ | $S_d$ | $R_o$ | $S_o$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $q_{av}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | DGA | 26.14 | 83.34 | 2602 | 2.51 | 12.74 | 11.48 | 11.9 | 62.9 | 6.2 | 9.58 | 56073 | 8884 |
| | OSPF | 36.75 | 48.3 | 2611 | 0.1 | 2.19 | 0 | 4 | 9 | 9 | 6.11 | - | - |
| | AntNet-CL | 43.5 | 48.7 | 2613 | 0.89 | 9.8 | 1.52 | 6 | 22.6 | 3 | 8.17 | 95167 | 15178 |
| | AntNet-CO | 43.26 | 49.8 | 2613 | 0.96 | 9.79 | 1.43 | 6.2 | 22.6 | 3 | 8.19 | 103831 | 15220 |
| | BeeHive | 45 | 38.43 | 2616 | 0.29 | 8.2 | 3.02 | 9 | 24.6 | 2.01 | 7.87 | 92375 | 11974 |
| | Daemon | 46.86 | 26.43 | 2622 | - | 3.38 | 0 | 4 | 10 | 0.85 | 6.87 | - | - |
| 100 | DGA | 27.84 | 147.1 | 2608 | 1.7 | 14.51 | 12.75 | 18.6 | 112.4 | 13.12 | 10.03 | 54950 | 8602 |
| | OSPF | 36.92 | 83.2 | 2612 | 0.1 | 2.21 | 0 | 4 | 9 | 16 | 6.13 | - | - |
| | AntNet-CL | 43.96 | 69.7 | 2621 | 0.89 | 10.41 | 1.74 | 8 | 36.8 | 5.8 | 8.31 | 97459 | 15200 |
| | AntNet-CO | 43.79 | 70.6 | 2623 | 0.95 | 10.46 | 1.76 | 8 | 37.2 | 5.8 | 8.32 | 105144 | 15230 |
| | BeeHive | 45.25 | 48.47 | 2622 | 0.29 | 8.3 | 3.15 | 11 | 38.3 | 3.3 | 7.88 | 75369 | 11970 |
| | Daemon | 46.92 | 28.59 | 2624 | - | 3.43 | 0 | 4 | 10 | 1.16 | 6.88 | - | - |
| 200 | DGA | 29.49 | 268.39 | 2627 | 1.06 | 15.81 | 13.57 | 29.5 | 201.8 | 26.85 | 10.25 | 56699 | 8422 |
| | OSPF | 37.15 | 154.6 | 2616 | 0.1 | 2.26 | 0 | 4 | 9 | 30.9 | 6.16 | - | - |
| | AntNet-CL | 44.7 | 106.1 | 2641 | 0.89 | 11.29 | 2.01 | 11.9 | 65.3 | 9.8 | 8.5 | 100047 | 15246 |
| | AntNet-CO | 44.6 | 108.2 | 2642 | 0.93 | 11.36 | 2.06 | 12 | 64.4 | 9.9 | 8.54 | 106533 | 15229 |
| | BeeHive | 45.93 | 66.46 | 2637 | 0.29 | 8.63 | 3.47 | 14.9 | 63.9 | 5.56 | 7.92 | 63308 | 11917 |
| | Daemon | 46.99 | 32.48 | 2627 | - | 3.57 | 0 | 4 | 11 | 1.71 | 6.91 | - | - |
| 300 | DGA | 30.28 | 380.92 | 2647 | 0.78 | 16.49 | 13.97 | 40.9 | 292.4 | 39.66 | 10.38 | 56091 | 8360 |
| | OSPF | 37.33 | 227.5 | 2621 | 0.1 | 2.28 | 0 | 4 | 9 | 45.8 | 6.18 | - | - |
| | AntNet-CL | 45.19 | 134.7 | 2664 | 0.89 | 11.79 | 2.18 | 14.9 | 87.3 | 13 | 8.61 | 101292 | 15285 |
| | AntNet-CO | 45.03 | 137.2 | 2666 | 0.92 | 11.89 | 2.29 | 15.2 | 88.6 | 13.2 | 8.65 | 107807 | 15235 |
| | BeeHive | 46.18 | 81.21 | 2655 | 0.29 | 8.74 | 3.48 | 18 | 86.3 | 7.41 | 7.95 | 56092 | 11893 |
| | Daemon | 47.03 | 36.46 | 2631 | - | 3.84 | 0 | 5 | 11 | 2.24 | 6.97 | - | - |
| 500 | DGA | 31.75 | 601.35 | 2701 | 0.52 | 17.26 | 14.24 | 60.1 | 456.3 | 64.92 | 10.44 | 54618 | 8318 |
| | OSPF | 37.59 | 377 | 2635 | 0.1 | 2.32 | 0 | 4 | 9 | 75.7 | 6.2 | - | - |
| | AntNet-CL | 45.82 | 177.4 | 2712 | 0.89 | 12.33 | 2.37 | 19.7 | 125.8 | 17.9 | 8.71 | 102764 | 15275 |
| | AntNet-CO | 45.8 | 179.8 | 2715 | 0.91 | 12.41 | 2.53 | 19.9 | 125.1 | 17.9 | 8.74 | 107597 | 15215 |
| | BeeHive | 46.45 | 104.33 | 2686 | 0.29 | 8.98 | 3.66 | 22.8 | 123.3 | 10.26 | 8 | 42240 | 11855 |
| | Daemon | 47.07 | 42.75 | 2639 | - | 3.89 | 0 | 5 | 12 | 3.13 | 6.98 | - | - |
| 1000 | DGA | 34.23 | 1078.9 | 2883 | 0.31 | 18.1 | 15.03 | 99.3 | 747.1 | 121.19 | 10.45 | 53018 | 8193 |
| | OSPF | 37.96 | 751.3 | 2677 | 0.1 | 2.35 | 0 | 4 | 9.9 | 149.8 | 6.21 | - | - |
| | AntNet-CL | 46.36 | 235.4 | 2824 | 0.89 | 12.82 | 2.65 | 26.6 | 182.9 | 23.9 | 8.82 | 104088 | 15230 |
| | AntNet-CO | 46.3 | 235.7 | 2825 | 0.91 | 12.88 | 2.76 | 26.1 | 182.9 | 23.9 | 8.83 | 107259 | 15135 |
| | BeeHive | 46.42 | 125.26 | 2755 | 0.29 | 9.12 | 3.79 | 27.9 | 162.5 | 12.67 | 8.06 | 29752 | 11705 |
| | Daemon | 47.1 | 45.83 | 2647 | - | 3.93 | 0 | 5 | 13 | 3.56 | 6.99 | - | - |
| 2500 | DGA | 36.64 | 1708.99 | 3533 | 0.19 | 16.34 | 14.63 | 128 | 946.2 | 221.17 | 9.74 | 63386 | 8148 |
| | OSPF | 38.3 | 1869.9 | 2808 | 0.1 | 2.38 | 0 | 4 | 10 | 366.8 | 6.22 | - | - |
| | AntNet-CL | 46.72 | 290.1 | 2988 | 0.9 | 13.09 | 2.77 | 33.3 | 247.6 | 29.8 | 8.86 | 103685 | 15143 |
| | AntNet-CO | 46.7 | 276.5 | 2973 | 0.9 | 13.21 | 2.93 | 31 | 234.3 | 28.2 | 8.89 | 106544 | 15042 |
| | BeeHive | 46.58 | 152.38 | 2846 | 0.29 | 9.24 | 4 | 33.4 | 219.3 | 15.98 | 8.08 | 26875 | 11605 |
| | Daemon | 47.1 | 45.92 | 2649 | - | 3.93 | 0 | 5 | 13 | 3.57 | 6.99 | - | - |
| 4000 | DGA | 37.03 | 1949.31 | 3958 | 0.16 | 14.86 | 13.26 | 122.8 | 943.8 | 272.37 | 9.31 | 73649 | 8162 |
| | OSPF | 38.41 | 2960.8 | 2934 | 0.1 | 2.39 | 0 | 4 | 10 | 577.1 | 6.22 | - | - |
| | AntNet-CL | 46.92 | 289.3 | 3012 | 0.9 | 13.23 | 2.85 | 33.5 | 247.1 | 29.6 | 8.88 | 104007 | 15124 |
| | AntNet-CO | 46.87 | 289.3 | 3009 | 0.9 | 13.26 | 2.94 | 32.4 | 244.7 | 29.7 | 8.9 | 106661 | 14993 |
| | BeeHive | 46.95 | 151.38 | 2855 | 0.29 | 9.18 | 3.87 | 33.1 | 221.5 | 15.92 | 8.04 | 26719 | 11562 |
| | Daemon | 47.1 | 45.92 | 2649 | - | 3.93 | 0 | 5 | 13 | 3.57 | 6.99 | - | - |

Table 3.9: Performance parameters for different buffer capacities in NTTNet

**Node150**

We decided to skip the buffer capacity experiments for Node150 network because experiments conducted on NTTNet network provided a good insight into this behavior. As we have already discussed small buffer capacities create a more relevant problem on mobile devices like PDAs. All modern routers can easily support a queue length of 1000 or more packets because of advancements in VLSI technology that lead to a cost-effective production of main memory chips.

### 3.9.3 Hot spots

The purpose of these experiments was to study the behavior of the algorithms in scenarios in which one node starts attracting bursts of traffic for a short period of time. The situation is quite common when some broadcasting channel breaks a news and everybody starts accessing its website then this website is acting as a hot spot in the network. A good dynamic routing algorithm should be able to manage and cope with hot spots in the networks.

**simpleNet**

All algorithms were able to cope with the hot spot traffic in simpleNet, therefore, we are skipping the results for the sake of brevity.

**NTTNet**

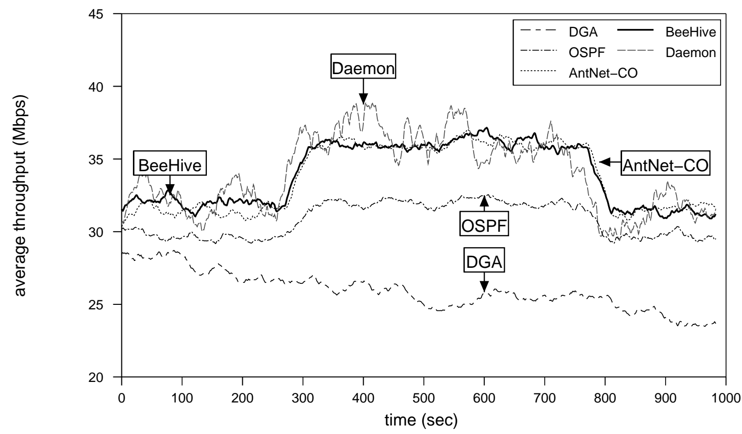
We made node 0 in Figure 3.6 to act as a hot spot from 300 seconds to 800 seconds which means that all nodes in the network sent data packets to this node with MPIA = 0.04 sec. This hot spot traffic was superimposed on a normal network traffic of MSIA = 3.6 sec, MPIA = 0.005 sec, sessionSize = 2130000 bits. The other parameters are $\delta_l = 512$ bytes and $\beta_c = 1000$ packets. Figure 3.22 summarizes the results. It is clear from the figure that *BeeHive*, *AntNet* and *Daemon* algorithms are able to cope with additional network traffic resulting from hot spot traffic. However, the packet delay of *BeeHive* is 50% less than that of *AntNet*. *OSPF* and *DGA* are unable to cope with hot spot traffic yet *DGA* has the worst performance among all algorithms. The other important parameters from the experiments are collected in Table 3.10. One can easily conclude that in almost all performance aspects *BeeHive* is better than *AntNet*. *Daemon*, as expected, has the best performance parameters (due to its access to the global network state).

**Node150**

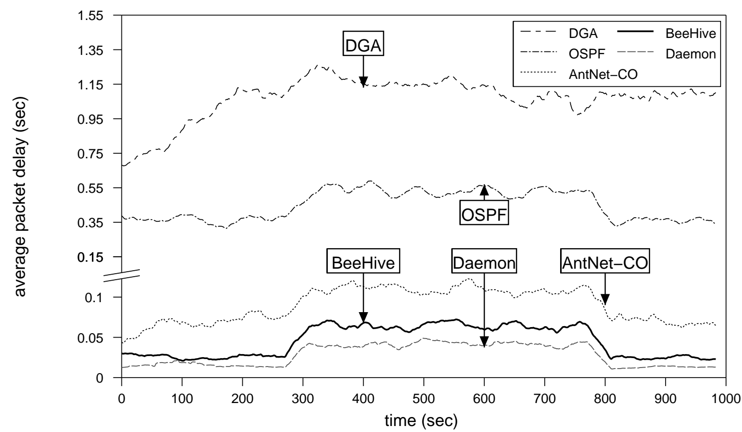
All input parameters for hot spot experiments on Node150 remained the same as in the experiments conducted on NTTNet except for the start and end time. In Nod150 the hot spot (node 0) was active from 500 seconds to 1000 seconds. From Figure 3.23 it is evident that *BeeHive* is able to maintain higher throughput and a significantly smaller packet delay than *AntNet* throughout the experiment. Please note that the time during which the hot spot is not active, packet delay of *AntNet* is approximately 20 msec greater than *OSPF*. The reason is that at MSIA = 3.6 sec no significant congestion resulted in the Node150 network. As a result, distributing packets on all possible paths, as *AntNet* does, is not a promising approach. We will explain this behavior of *AntNet* later under the session-less traffic experiments. As expected, *BeeHive* and *AntNet* are again able to cope better with the hot spot traffic as compared with *OSPF*. The other important parameters are collected in Table 3.10. Please note that at this relatively bigger topology, performance parameters of *BeeHive* are significantly better than *AntNet*. The reason for it was already explained during the discussion of congestion control experiments.

### 3.9.4 Router crash experiments

The purpose of these experiments is to study the fault tolerant behavior of different algorithms. The experiments provide an insight into how quickly an algorithm adapts its routes if a router

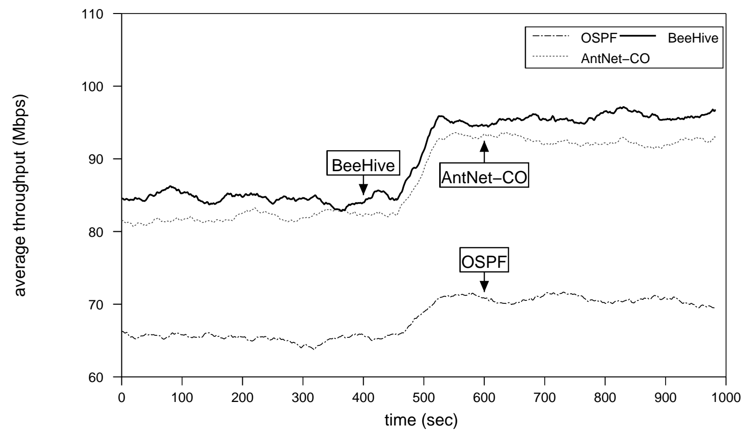



(a) Average throughput

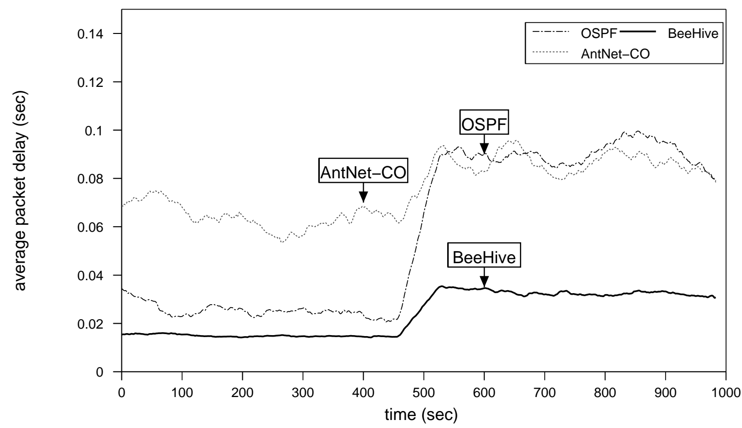



(b) Average packet delay

Figure 3.22: Hot spot is Node 0 in NTTNet




(a) Average throughput




(b) Average packet delay

Figure 3.23: Hot spot is Node 0 in Node150

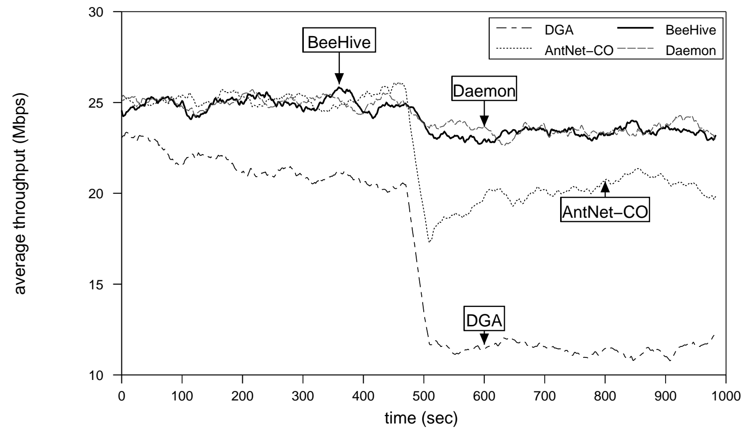
| Experiment | Algorithm | $P_d$ | $S_c$ | $S_d$ | $S_{90d}$ | $R_o$ | $S_o$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NTTNetHot | DGA | 74.17 | 39.08 | 2865 | 3570 | 0.4 | 16.75 | 16.38 | 95.19 | 11.43 | 49484 | 8121 |
| | OSPF | 87.96 | 57.04 | 2721 | 3098 | 0.1 | 2.44 | 0 | 118.4 | 6.75 | - | - |
| | AntNet-CL | 98.06 | 93.18 | 2724 | 3052 | 1.71 | 20.07 | 2.51 | 19.9 | 9 | 105139 | 14740 |
| | AntNet-CO | 97.97 | 93.22 | 2727 | 3063 | 1.76 | 20.38 | 2.62 | 20.9 | 9.05 | 109717 | 14687 |
| | BeeHive | 98.29 | 96.09 | 2667 | 2898 | 0.27 | 7.09 | 2.98 | 10.84 | 8.21 | 31181 | 11632 |
| | Daemon | 98.22 | 97.11 | 2624 | 2777 | - | 2.43 | 0 | 8.63 | 6.98 | - | - |
| Node150Hot | OSPF | 95.19 | 89.69 | 2640 | 2825 | 0.2 | 0.84 | 0 | 5.47 | 5.4 | - | - |
| | AntNet-CO | 98.66 | 94.43 | 2709 | 3203 | 2.45 | 10.7 | 2.19 | 14.6 | 8.53 | 112412 | 22553 |
| | BeeHive | 99.55 | 97.73 | 2633 | 2803 | 0.73 | 2.17 | 0.62 | 7.54 | 5.77 | 36148 | 13606 |
| NTTNetExp1 | DGA | 67.13 | 39.12 | 2795 | 3434 | 0.41 | 10.75 | 14.84 | 76.96 | 11.34 | 47548 | 7733 |
| | AntNet-CL | 92.02 | 79.54 | 2679 | 2960 | 1.63 | 10.76 | 2.12 | 9.1 | 8.21 | 82438 | 14572 |
| | AntNet-CO | 92.71 | 79.58 | 2690 | 2994 | 1.63 | 10.98 | 2.41 | 11.4 | 8.25 | 85599 | 14537 |
| | BeeHive | 99.67 | 92.48 | 2669 | 2960 | 0.22 | 5.02 | 7.19 | 6.03 | 8.09 | 27003 | 11447 |
| | Daemon | 100 | 96.14 | 2620 | 2771 | - | 1.35 | 0 | 0.39 | 6.68 | - | - |
| NTTNetExp2 | DGA | 64.57 | 39.07 | 2800 | 3455 | 0.3 | 9.32 | 12.82 | 85.77 | 10 | 45750 | 7628 |
| | AntNet-CL | 93.73 | 81.44 | 2686 | 2977 | 1.65 | 11.46 | 2.39 | 9.5 | 8.32 | 82947 | 14630 |
| | AntNet-CO | 94.17 | 81.54 | 2686 | 2974 | 1.68 | 11.29 | 2.59 | 9.6 | 8.3 | 85699 | 14557 |
| | BeeHive | 99.7 | 93.15 | 2669 | 2962 | 0.22 | 5.11 | 7.46 | 5.83 | 8.11 | 26998 | 11592 |
| | Daemon | 100 | 96.84 | 2621 | 2772 | - | 1.32 | 0 | 0.43 | 6.66 | - | - |

Table 3.10: Performance parameters for hot spot and router down experiments

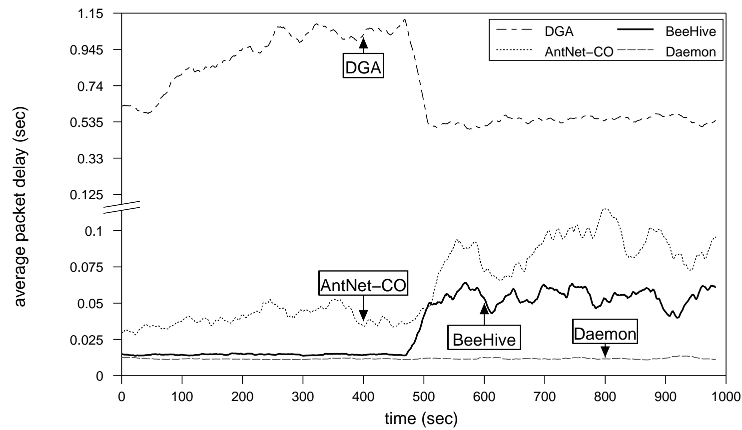
crashes. Two features are of primary importance to handle fault tolerance in networks: one, a routing algorithm should be able to reroute packets on alternate paths toward their destination when an existing path is no more available, and two, once a router is repaired then its routing table should adapt quickly in order to start routing packets as quickly as possible. We do not report experiments for simpleNet, as one can not see a significant difference in performance among different algorithms and consequently we lack the resources to complete all experiments on Node150, therefore, we do only important experiments on Node150. We dropped *OSPF* from the fault tolerant experiments because it has significantly poor performance as compared to *AntNet* and *BeeHive* without any router crash. We are including *DGA* for the sake of completeness otherwise we could have dropped it easily because it performed worst among all algorithms.

### NTTNet

We report two experiments in which we analyzed the fault tolerant behavior of the algorithms.
**Experiment1.** In Experiment 1 the traffic generator parameters were MSIA = 4.6 sec, MPIA = 0.005 sec, sessionSize = 2130000 bits, $\delta_l = 512$ bytes and $\beta_c = 1000$ packets. In this experiment Router 20 and Router 43 (see Figure 3.6) both crashed at 500 seconds and then remained down for the rest of the experiment. From Figure 3.6 it is clear that router 20 and 43 are critical routers in NTTNet. One can easily conclude from Figure 3.24 that *BeeHive* and *Daemon* are able to maintain significantly higher throughput as compared to *AntNet*. The packet delay of *Daemon* is significantly smaller as compared to *BeeHive* and *AntNet*, however, *BeeHive* is able to maintain approximately half of the packet delay as compared to *AntNet*. We have collected other important parameters in Table 3.10. Please note that all performance parameters of *BeeHive* are significantly better than *AntNet*. Packet delivery ratio of *BeeHive* is approximately the same as that of *Daemon*, which shows that *bee agents* were able to adapt routes once the routers crashed and *AntNet*, as expected, is not able to cope with the router crash problem.
**Experiment2.** All of the traffic generator's parameters were the same in this experiment as in the previous experiment except the router crash times. In this experiment, Router 20 crashed at 300 seconds, and Router 43 crashed at 500 seconds and both routers were repaired at 800 seconds. The throughput and packet delay of algorithms are shown in Figure 3.25. The tendency of the algorithms is quite similar to the one as shown in Figure 3.24. *BeeHive* is able to maintain higher throughput and lower packet delay as compared to *AntNet*. All other important parameters are collected in Table 3.10. *Daemon* is the best performing algorithm among all algorithms while *DGA* is the worst performing algorithm. Please note that once Router 43 crashes, multiple paths

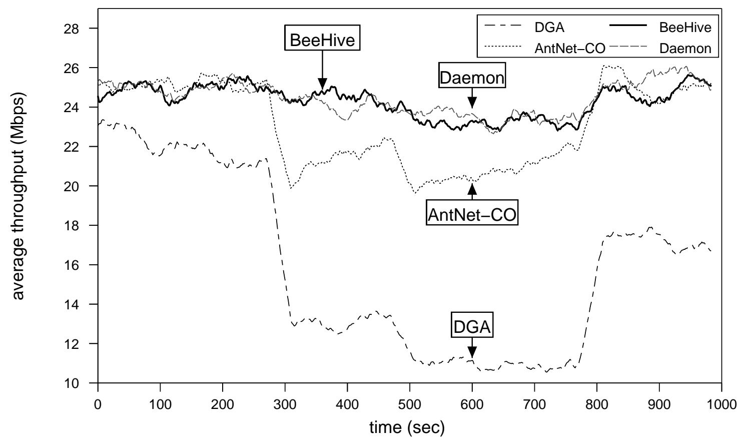



(a) Average throughput

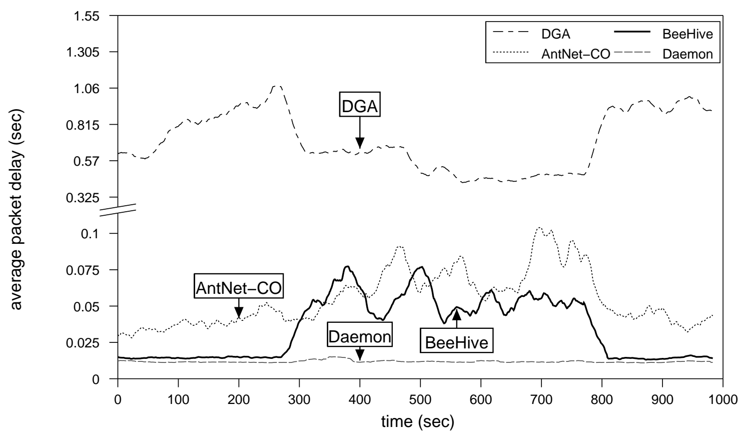



(b) Average packet delay

Figure 3.24: Router 20 and Router 43 crashed at time = 500 seconds




(a) Average throughput




(b) Average packet delay

Figure 3.25: Router 20 crashed at 300 seconds and Router 43 crashed at 500 seconds and both were repaired at 800 seconds

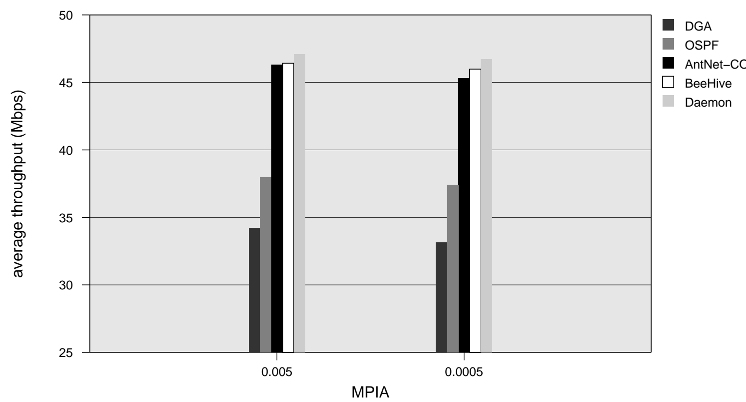
still exist via Router 40 to the upper part of the network topology therefore *BeeHive* is able to deliver more packets than *AntNet*.

### 3.9.5 Bursty traffic generator

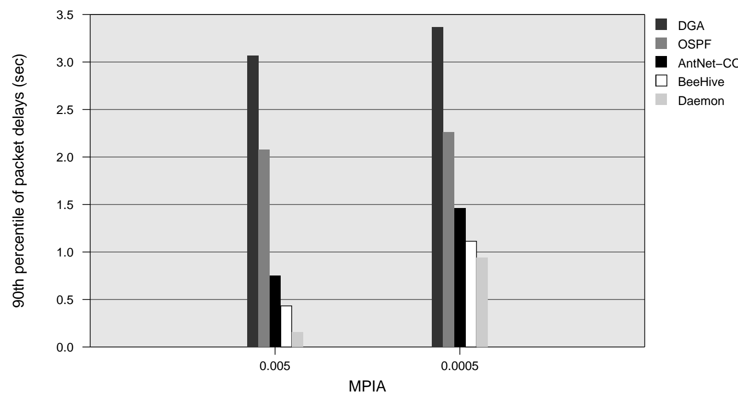
The purpose of these set of experiments was to study the behavior of the algorithms under bursty network traffic. Such traffic consists of sudden bursts of packets followed by a long period of silence/inactivity (no traffic). Such a scenario is important because it investigates how quickly an algorithm can react to changes in network traffic. We skip the results for simpleNet as the performance of all algorithms were approximately the same for bursty traffic.

#### NTTNet

One can generate bursty traffic patterns using our traffic generator by decreasing the value of MPIA. In the experiments we decreased the value of MPIA from 0.005 sec to 0.0005 sec, which made a session to finish in one tenth of the time (260 msec) as compared to at MPIA = 0.005 sec. However, we kept the value of MSIA constant at 2.6 sec. This resulted in sending 2130000 bits (size of a session) in 260 msec followed by an inactivity period of approximately 2 seconds. Other parameters were $\delta_l = 512$ bytes and $\beta_c = 1000$ packets. One can see in Figure 3.26 that all




(a) Average throughput




(b) 90th percentile of packet delays

Figure 3.26: Bursty traffic behavior in NTTNet

algorithms are able to deliver approximately the same number of packets at MPIA = 0.0005 sec

| MPIA | Algorithm | $P_d$ | $t_d$ | $S_c$ | $S_d$ | $S_{90d}$ | $R_o$ | $S_o$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.005 | DGA | 72.81 | 1079 | 31.19 | 2883 | 3559 | 0.31 | 18.1 | 15.03 | 99.3 | 747.1 | 10.45 | 53028 | 8183 |
| | OSPF | 80.86 | 751 | 48.39 | 2677 | 2962 | 0.1 | 2.35 | 0 | 4 | 9.9 | 6.21 | - | - |
| | AntNet-CL | 99.17 | 235 | 88.18 | 2824 | 3303 | 0.89 | 12.82 | 2.65 | 26.6 | 182.9 | 8.82 | 104061 | 15220 |
| | AntNet-CO | 99.2 | 236 | 88.68 | 2825 | 3310 | 0.91 | 12.88 | 2.76 | 26.1 | 182.9 | 8.83 | 107924 | 15249 |
| | BeeHive | 99.69 | 125 | 93.78 | 2755 | 3143 | 0.29 | 9.12 | 3.79 | 27.9 | 162.5 | 8.06 | 29733 | 11702 |
| | Daemon | 99.99 | 46 | 99.51 | 2647 | 2836 | - | 3.93 | 0 | 5 | 13 | 6.99 | - | - |
| 0.0005 | DGA | 70.91 | 1322 | 37.23 | 1019 | 1901 | 0.28 | 16.5 | 14.19 | 103.1 | 772.8 | 10.14 | 52567 | 7562 |
| | OSPF | 79.92 | 910 | 51.01 | 721 | 1297 | 0.1 | 2.34 | 0 | 1 | 3 | 6.23 | - | - |
| | AntNet-CL | 96.68 | 655 | 76.39 | 1113 | 1968 | 0.77 | 12.4 | 3.02 | 62.4 | 358.6 | 8.77 | 111788 | 14600 |
| | AntNet-CO | 97.26 | 634 | 79.66 | 1111 | 1967 | 0.83 | 12.74 | 3.21 | 59.5 | 343.1 | 8.82 | 112325 | 14592 |
| | BeeHive | 98 | 474 | 83 | 989 | 1714 | 0.29 | 9.28 | 4.19 | 86.2 | 372.1 | 8.08 | 31041 | 10255 |
| | Daemon | 99.58 | 447 | 96.92 | 883 | 1488 | - | 13.18 | 0 | 14 | 75 | 8.86 | - | - |

Table 3.11: Performance parameters for bursty traffic generators on NTTNet

| MPIA | Algorithm | $P_d$ | $t_d$ | $S_c$ | $S_d$ | $S_{90d}$ | $R_o$ | $S_o$ | $P_{loop}$ | $J_d$ | $J_{90d}$ | $h_{av}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.005 | OSPF | 93.23 | 236 | 68.08 | 2663 | 2888 | 0.2 | 1.26 | 0 | 4 | 9.2 | 5.34 | - | - |
| | AntNet-CO | 99.16 | 109 | 95.89 | 2710 | 3134 | 2.36 | 12.42 | 1.73 | 14.6 | 146.6 | 8.12 | 112666 | 23120 |
| | BeeHive | 99.94 | 37 | 98.45 | 2642 | 2817 | 0.81 | 2.37 | 0.6 | 7 | 35.8 | 5.62 | 34399 | 13399 |
| 0.0005 | OSPF | 90.47 | 393 | 69.71 | 628 | 1002 | 0.2 | 1.21 | 0 | 1 | 2 | 5.32 | - | - |
| | AntNet-CO | 98.18 | 410 | 85.41 | 921 | 1631 | 2.25 | 12.62 | 1.97 | 43.2 | 291 | 8.21 | 117440 | 22950 |
| | BeeHive | 99 | 218 | 91.44 | 651 | 999 | 0.81 | 2.5 | 0.7 | 22.6 | 129 | 5.66 | 36870 | 11862 |

Table 3.12: Performance parameters for bursty traffic generators on Node150

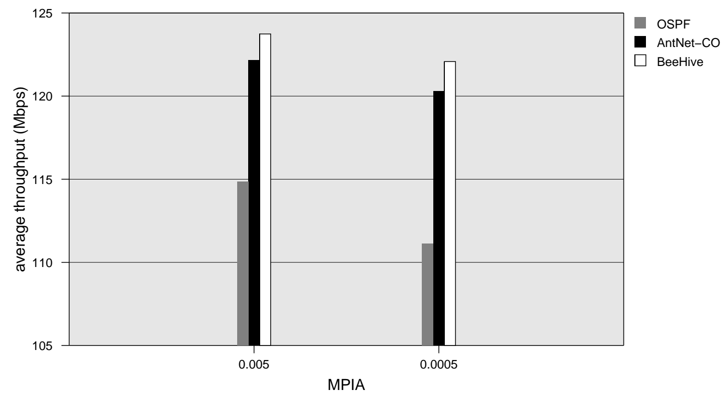
as compared to MPIA = 0.005 sec but with a significantly greater packet delay. The packet delay of *BeeHive* is quite close to *Daemon* and significantly lower than *AntNet*. All other important parameters are collected in Table 3.11. Please note that all performance parameters of *BeeHive* are significantly better than that of *AntNet*. We again see the same tendency that *Daemon* is the best performing algorithm while *DGA* is the worst performing algorithm. The better performance of *BeeHive* is due to its quick spreading of routing information by utilizing only forward moving agents (see Section 3.4).

**Node150**

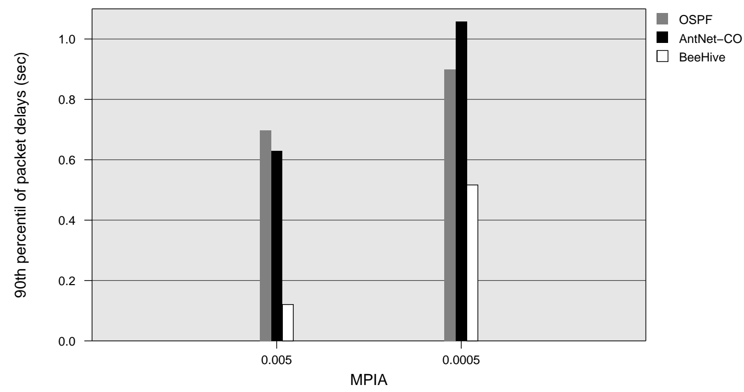
We continued with the bursty traffic experiments on Node150. The traffic generator at a node received the same parameters as that of the previous experiment. The throughput and 90th percentile of packet delays are shown in Figure 3.27. *BeeHive* is able to maintain significantly higher throughput and lower packet delay as compared to *AntNet*. All other important parameters are collected in Table 3.12. Please note that the advantage of quick communication-oriented spreading of the routing information is now more visible as the performance values of *BeeHive* are an order of magnitude better than that of *AntNet*. Please note that *BeeHive* has a session delay of 1000 msec, suboptimal overhead of 2.5%, average hop count of 5.6 as compared to 1631 msec, 12.6% and 8.2 hops for *AntNet*, respectively.

### 3.9.6 Session-less network traffic

The purpose of these experiments was twofold, first to test the algorithms in a domain where the performance of *OSPF* is the best, and second, to be confident that our implementation of *DGA* is functionally correct. The authors of *DGA* only published their results under a session-less network traffic with MPIA = 0.035 sec. The poor performance of *DGA* at higher loads, reported in the previous sections, required us to verify its functional correctness. One option is to use the same traffic patterns that the developers of *DGA* utilized and then compare the results. The improvements that we made in *DGA* to make it competitive with other algorithms might have introduced some undesired side effects. However, our results form such extensive studies further strengthened our belief that our improved version of *DGA* was still functionally similar to that




(a) Average throughput



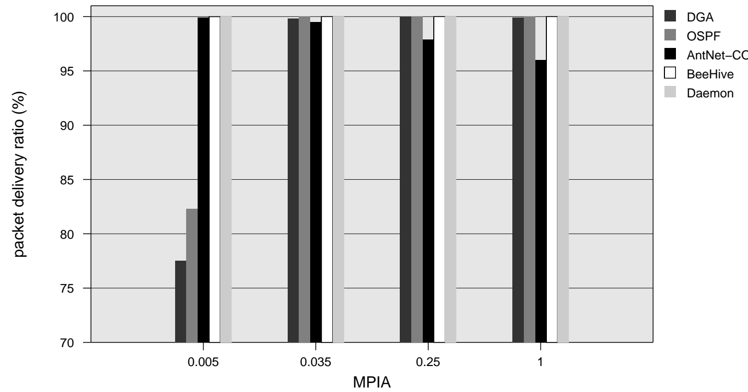

(b) 90th percentile of packet delays

Figure 3.27: Bursty traffic behavior in Node150

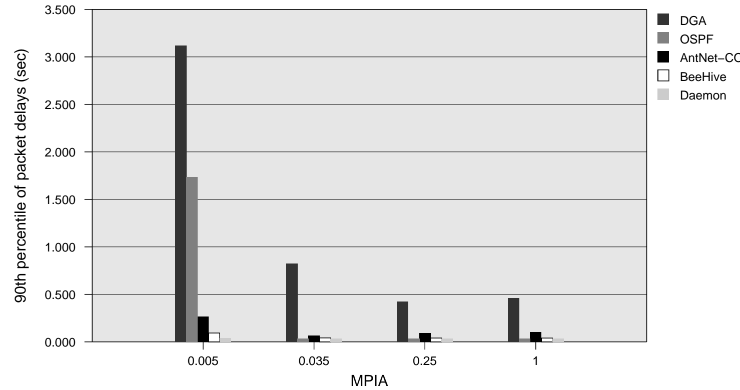
of original *DGA*. We skip reporting the results for simpleNet for the sake of brevity because the results do not show any significant difference in performance among the algorithms.

**NTTNet**

We use the session-less traffic generator in which the destination for each packet is chosen at random from a uniform distribution. We gradually decreased the load to a static level by increasing the MPIA from 0.005 sec to 1 sec. The idea is to test the feasibility of the algorithms under static conditions where *OSPF* is a state-of-the-art algorithm. At MPIA = 0.005 sec, i.e. under a significantly high load, *BeeHive* and *AntNet* completely outperformed *DGA* and *OSPF* but as the load started to decrease, the packet delivery ratio of *OSPF* (see Figure 3.28) significantly improved. Please note that *AntNet* drops about 2-3% packets at low loads. This came as a surprise to us. We investigated the problem and it appeared that stochastic distribution of packets over too many multiple paths, a desired property under high network traffic, is not a promising approach under static network loads. As a result more data packets followed loops as shown by the $p_{loop}$ parameter in Table 3.13 and the packets that were dropped were those that have followed about 100 hops and had not arrived at the destination. The results in Figure 3.28 clearly




(a) Packet delivery ratio




(b) 90th percentile of packet delays

Figure 3.28: Session-less network traffic for NTTNet

demonstrate the superiority of *OSPF* over all other algorithms under low loads. Please closely monitor the performance values of *BeeHive* in comparison to *OSPF*. It appears that maintaining only those paths towards a destination whose quality is above a threshold value (bee behavior)

| MPIA | Algorithm | $T_{av}$ | $t_d$ | $R_o$ | $S_o$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $P_{drop}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.005 | DGA | 36.23 | 1046.57 | 0.31 | 23.21 | 17.37 | 112.55 | 11.65 | 22.33 | 48351 | 6913 |
| | OSPF | 38.46 | 627.6 | 0.1 | 2.4 | 0 | 112 | 6.23 | 17.66 | - | - |
| | AntNet-CL | 46.74 | 35.5 | 0.79 | 9.83 | 1.08 | 1 | 8.19 | 0.01 | 113576 | 15599 |
| | AntNet-CO | 46.74 | 35.3 | 0.78 | 9.45 | 1 | 1 | 8.11 | 0.01 | 118669 | 14525 |
| | BeeHive | 46.74 | 33.78 | 0.3 | 8.83 | 3.23 | 1.32 | 7.98 | 0.01 | 25614 | 11268 |
| | Daemon | 46.74 | 22.03 | - | 2.44 | 0 | 0.27 | 6.66 | 0 | - | - |
| 0.035 | DGA | 6.67 | 150.64 | 4.39 | 14.1 | 36.95 | 2.08 | 26.69 | 0.17 | 51028 | 9345 |
| | OSPF | 6.68 | 20 | 0.1 | 0.51 | 0 | 10 | 6.9 | 0 | - | - |
| | AntNet-CL | 6.68 | 24 | 0.75 | 1.27 | 0.93 | 0 | 8 | 0.02 | 116385 | 15868 |
| | AntNet-CO | 6.68 | 24 | 0.75 | 1.19 | 0.87 | 0 | 7.88 | 0.02 | 123341 | 15132 |
| | BeeHive | 6.68 | 23.03 | 0.19 | 1.16 | 2.66 | 0.01 | 7.84 | 0 | 25046 | 11144 |
| | Daemon | 6.68 | 19.83 | - | 0.29 | 0 | 0.01 | 6.58 | 0 | - | - |
| 0.25 | DGA | 0.93 | 87.67 | 4.98 | 2.17 | 35.25 | 0.04 | 28.67 | 0.01 | 51629 | 12036 |
| | OSPF | 0.94 | 20 | 0.1 | 0.07 | 0 | 12 | 6.9 | 0 | - | - |
| | AntNet-CL | 0.93 | 25.5 | 0.74 | 0.24 | 3.15 | 0 | 8.6 | 0.11 | 115419 | 16285 |
| | AntNet-CO | 0.93 | 25.3 | 0.74 | 0.23 | 3.15 | 0 | 8.5 | 0.11 | 122168 | 16097 |
| | BeeHive | 0.93 | 22.48 | 0.17 | 0.15 | 1.76 | 0 | 7.66 | 0 | 25409 | 11130 |
| | Daemon | 0.94 | 19.76 | - | 0.04 | 0 | 0 | 6.58 | 0 | - | - |
| 1 | DGA | 0.23 | 86.45 | 5.04 | 0.54 | 32 | 0.02 | 28.56 | 0.06 | 49657 | 13465 |
| | OSPF | 0.23 | 20 | 0.1 | 0.02 | 0 | 12 | 6.9 | 0 | - | - |
| | AntNet-CL | 0.23 | 29.9 | 0.72 | 0.1 | 8.77 | 0 | 10.08 | 0.33 | 116211 | 16990 |
| | AntNet-CO | 0.23 | 30 | 0.72 | 0.1 | 8.72 | 0 | 10 | 0.34 | 123457 | 16959 |
| | BeeHive | 0.23 | 22.09 | 0.17 | 0.04 | 1.21 | 0 | 7.5 | 0 | 25508 | 11178 |
| | Daemon | 0.23 | 19.74 | - | 0.01 | 0 | 0 | 6.58 | 0 | - | - |

Table 3.13: Performance parameters for Session-less traffic in NTTNet

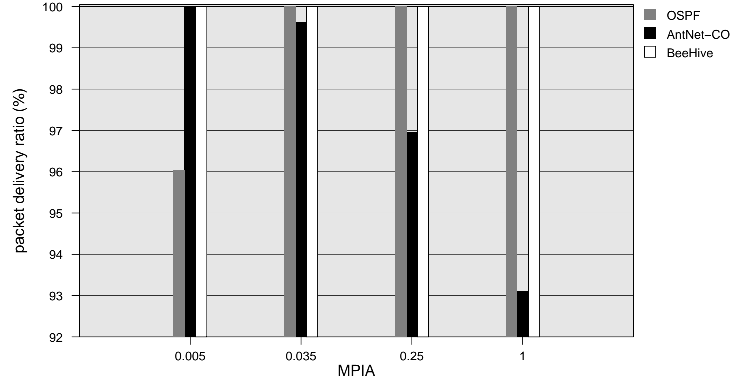
results in striking a good compromise between *AntNet*, which maintains all possible multiple paths towards a destination, and *OSPF* which maintains a single path towards the destination. All other important performance parameters are collected in Table 3.13. Our claim is verified by the performance parameters collected in Table 3.13. The performance of *BeeHive* is similar or better than that of *AntNet* at higher loads (MPIA = 0.005 sec) and similar to that of *OSPF* under low loads. Please compare the values of the $p_{loop}$ and $h_{av}$ parameters under a low network traffic load (MPIA = 1.0 sec).

**Node150**

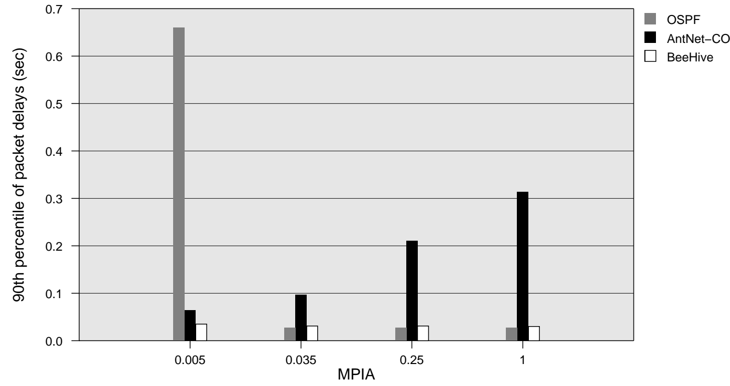
We used the same traffic generator parameters as in the previous experiment on Node150 network and the tendency of the performance parameters are approximately the same. It is clear from Figure 3.29 that *AntNet* now drops approximately 5-6% of packets and has a significantly higher packet delay as compared to *BeeHive* and *OSPF* under low loads. All other performance parameters are collected in Table 3.14. The performance values of *BeeHive* are again quite close to *AntNet* under high loads and quite close to *OSPF* under low loads. Now compare the $p_{loop}$ and $h_{av}$ values of all algorithms. One can see that about 19% packets enter into loops in *AntNet* as compared to 0.7% for *BeeHive* for MPIA = 1.0 sec. Consequently, the average hop count increases to 24 hops for *AntNet*. These results show that for large topologies and for extremely low network traffic, routing tables in *AntNet* do not converge, as a result, 5-6 % of packets keep on looping in the networks. We believe that a threshold-based exploitation of paths, such as the bee behavior, has been instrumental for *BeeHive* in getting a performance quite close to *OSPF* under low loads.

### 3.9.7 Size of routing table

The size of routing tables utilized by *AntNet*, *BeeHive* and *OSPF* is shown in Figure 3.30. The benefits of the *foraging zone* and *foraging region* concepts become clear for bigger and complex topologies. For NTTNet, *AntNet* has 162 entries on the average, in comparison to 78 and 57 for *BeeHive* and *OSPF* respectively. For Node150 *AntNet*, *BeeHive* and *OSPF* have 400, 194 and 150




(a) Packet delivery ratio




(b) 90th percentile of packet delays

Figure 3.29: Session-less network traffic for Node150

| MPIA | Algorithm | $T_{av}$ | $t_d$ | $R_o$ | $S_o$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $P_{drop}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.005 | OSPF | 118.13 | 224.4 | 0.2 | 1.33 | 0 | 73 | 5.36 | 3.95 | - | - |
| | AntNet-CO | 122.98 | 31 | 2.09 | 10.57 | 1.14 | 0 | 7.65 | 0.02 | 119621 | 24701 |
| | BeeHive | 123.03 | 22.64 | 0.81 | 2.7 | 0.91 | 0.32 | 5.7 | 0 | 31311 | 13813 |
| 0.035 | OSPF | 17.57 | 19 | 0.2 | 0.21 | 0 | 8 | 5.4 | 0 | - | - |
| | AntNet-CO | 17.5 | 33 | 2.08 | 2.03 | 2.14 | 0 | 8.57 | 0.38 | 116840 | 27320 |
| | BeeHive | 17.57 | 20.76 | 0.5 | 0.38 | 0.89 | 0.01 | 5.69 | 0 | 31029 | 14422 |
| 0.25 | OSPF | 2.46 | 19 | 0.2 | 0.03 | 0 | 8 | 5.4 | 0 | - | - |
| | AntNet-CO | 2.39 | 55.2 | 2.05 | 0.74 | 8.36 | 0 | 14.47 | 3.03 | 114867 | 29997 |
| | BeeHive | 2.46 | 20.49 | 0.46 | 0.05 | 0.78 | 0 | 5.65 | 0 | 31102 | 14366 |
| 1 | OSPF | 0.62 | 18.8 | 0.2 | 0.01 | 0 | 8 | 5.4 | 0 | - | - |
| | AntNet-CO | 0.57 | 91.2 | 2 | 0.36 | 19.16 | 0 | 23.87 | 6.89 | 112407 | 31462 |
| | BeeHive | 0.62 | 20.33 | 0.45 | 0.01 | 0.7 | 0 | 5.61 | 0 | 31421 | 14830 |

Table 3.14: Performance parameters for Session-less traffic in Node150

entries respectively in the routing table. The number of entries for *BeeHive* are a sum of number of entries of all three routing tables maintained by *BeeHive*.
Figure 3.30 demonstrates the clear advantage of the way route discovery and maintenance is done in *BeeHive*. The concepts of *foraging zone* and *foraging region* not only result in smaller routing tables but also in a smaller routing overhead as compared to *AntNet*. We believe that for topologies larger than 150 nodes, *BeeHive* will maintain significantly smaller routing tables as compared to *AntNet*.

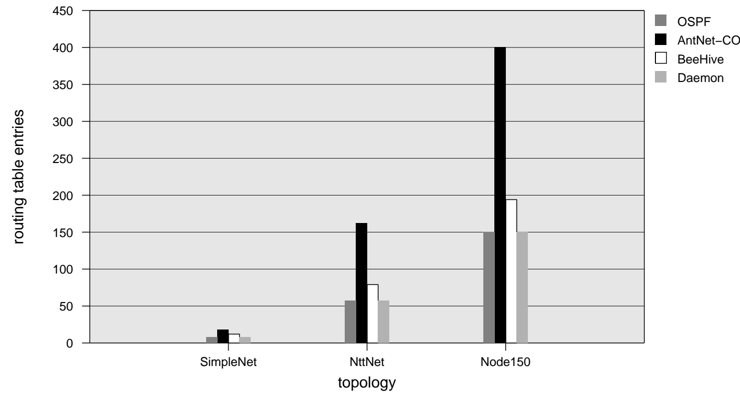



Figure 3.30: Size of routing table

## 3.10 Summary

A honey bee colony is able to optimize its stockpiles of nectar, pollen and water through an intelligent allocation of labor among different specialists, who communicate with each other using a sophisticated communication protocol that consists of *signals* and *cues*, in continuously changing internal and external environments. The dance language and foraging behavior of honey bees inspired us to develop a dynamic, simple, efficient, robust, flexible and scalable multi-path routing algorithm. The algorithm does not need any global information such as the structure of the topology and cost of links among routers, rather it works with the local information that a *short distance bee agent* collects in a *foraging zone*. *BeeHive* does not utilize an optimized clustering algorithm to avoid the overhead of routing through cluster heads. It works without the need of global clock synchronization which not only simplifies its installation on real routers but also enhances fault tolerance. In contrast to *AntNet* our algorithm utilizes only forward-moving *bee agents* that help in disseminating the state of the network to the routers in real-time. The *bee agents* take less than 1% of the available bandwidth but provide significant enhancements in throughput and packet delay over other state-of-the-art approaches.
We implemented two state-of-the-art Nature inspired algorithms (*AntNet* and *DGA*) for the OMNeT++ simulator and then compared our *BeeHive* algorithm with them. Through extensive simulations representing dynamically changing operating network environments we have demonstrated that *BeeHive* achieves a better or similar performance as compared to *AntNet*. However, this enhancement in performance is achieved with following preferred features:

- Significantly smaller routing tables which have the order of the size as in *OSPF*.
- Simple agents which resulted in significantly less control overhead as compared to *AntNet*.
- Simple agents also resulted in significantly less processing complexity as compared to *AntNet*.
- No infinite loop problems, which *AntNet* has, under static network traffic.

- Performance is competitive to *OSPF* under low loads.
- Better scalability to large topologies.
- Easier implementation in Linux routers than *AntNet* or other protocols.

# 4

# A Scalability Framework for Nature Inspired Routing Algorithms

*The major contribution of the work presented in this chapter is a comprehensive framework for the scalability analysis of distributed routing protocols. The framework models the productivity of a routing protocol on a number of performance values. The cost model consists of processing, communication and resource costs for deploying a routing algorithm in an operational environment in real world networks. The framework is general enough and can be easily utilized to investigate the scalability of an agent-based distributed software system. Finally, we studied the scalability behavior of two state-of-the-art Nature inspired agent-based routing protocols:* AntNet *and* BeeHive. *We also evaluated the scalability of* OSPF *which is currently employed in the Internet. The results give valuable insight into the scaling capacity of agent-based routing algorithms. We believe that the work will motivate designers of routing protocols to consider scalability as an important metric in the design and development of state-of-the-art routing protocols and to empirically validate them according to this measure.*

## 4.1 Introduction

A routing protocol has to be deployed on large scale telecommunication networks. Therefore, not envisaging the repercussions of scalability on the performance of a routing protocol might lead to severe performance bottlenecks. Consequently, the web applications, which utilize the networking systems running such protocols, might show poor responsiveness resulting in customer dissatisfaction and loss of valuable revenues in a highly competitive market [180, 179]. Scalability means the ability of a routing protocol to efficiently transport the network traffic between any pair of the nodes with adequate quality of service, over a wide range of network configurations. The increased performance should be in proportion to additional costs [103]: processing, communication and router's resources.

The major factors that might undermine the scalability of a routing protocol are: excessive consumption of router's resources and the routing complexity [242]. In order to counter these problems, designers of the routing protocols have to be equipped with analytical and empirical tools to systematically investigate the scalability behavior of a routing algorithm. In this chapter, we present a new scalability framework for a routing protocol, which has the following new features:

- It provides a comprehensive cost model which incorporates the processing, communication, and router's resource costs.
- It utilizes a recently proposed concept of *total overhead* in [165] for calculating the communication costs rather than the existing practice of taking only the control overhead.
- it provides a quality-of-service value which depends on a number of refined and processed parameters rather than only delay.

- it utilizes a refined throughput metric for a routing algorithm rather than the raw throughput.
- It defines power and productivity metrics for a routing algorithm, on the basis of above-mentioned parameters, which provide significant insight into its benefit-to-cost ratio.
- It defines a scalability matrix, which facilitates the scalability analysis. The matrix enables a designer to study the scalability of a routing algorithm either across different topologies by fixing the network traffic load, or on the same topology by increasing the network traffic load.
- It uses a new comprehensive empirical performance evaluation framework that collects the relevant performance values required for the scalability framework.

We believe that our new framework will enable the designers of routing protocols to establish the scalability of their routing protocol in an early stage of *protocol engineering* [113]. Such a framework will be instrumental in practicing the principles of *Software Performance Engineering (SPE)*, which also emphasizes the consideration of performance and scalability issues early in the design and architectural phase [180, 181, 236, 179], to rectify the deficiencies in a simulation environment. This will not only obviate the risk of a disaster once the algorithm is deployed on large scale networks but also avert the cost overruns due to tuning or redesigning the algorithm later in the protocol engineering cycle. Consequently, such a pragmatic protocol engineering cycle will be capable of reducing the time-to-market of a new protocol.
The framework is general enough to act as a guideline for analyzing the scalability of any agent-based network system. However, in this chapter, we limit our analysis to two state-of-the-art agent-based routing algorithms: *AntNet* and *BeeHive*. We will also analyze the scalability of a classical non-adaptive routing protocol *OSPF*.
We now briefly discuss the existing work on scalability analysis. We will point out considerable lack of concrete work on comprehensive scalability analysis for routing protocols. This should facilitate to emphasize our novel direction of work.

### 4.1.1 Existing work on the scalability analysis

The scalability analysis has received extensive treatment in the area of parallel computing. Here the focus is to analyze the scalability of a parallel algorithm on massively parallel platforms. The important metrics are: *speedup, efficiency and scalability* [193]. Speedup measures how the rate of doing work increases with an increase in the number of processors as compared to one processor. Efficiency is the work rate per processor, and scalability is defined as the ratio of efficiencies on two platforms. Ideally, efficiency and scalability metrics have a value of unity while speedup is k if k processors are added from one configuration to another. The interested reader will find the detailed treatment of the scalability of parallel computing systems and algorithms in [83, 87, 178, 115, 166].
The work on analyzing the scalability of distributed systems is inspired by the work of Giessler et al. [81], in which they proposed a *power metric* for computer network systems. The metric is defined as $P = \frac{T_{av}}{tr}$ where $T_{av}$ is the average throughput of the system and $t_r$ is the ratio of average packet delay to the minimum packet delay. Kleinrock extended this definition in [112] to combined loss and delay systems, and was able to define the optimal operating point at which the defined power is maximized. In [111], he discussed the effect of flow control procedures on the throughput in computer networks. Earlier work of Kuemmerle and Rudin presented in [116] is also of paramount importance because the authors compared the performance of circuit switched networks with packet switched networks, considering delay performance and usage cost. Our cost model utilizes some of the definitions presented in their work. In [159], Rosner concluded that packet switched networks outperform circuit switched networks with respect to both transmission utilization efficiencies and overall network costs. However, the major focus of these studies does not relate to comparing the efficiencies of routing protocols, which is the focus of our work.
In [101, 102, 103], Jogalekar and Woodsite, somehow misinterpreted $t_r$ (introduced in the last

paragraph), for the average packet delay. As a result, they thought that the power is significantly influenced by smaller packet delays. Therefore, they modified the definition of power for distributed systems as

$$P(k) = \frac{\lambda(k)}{1 + \frac{T(k)}{T'}} \tag{4.1}$$

where $\lambda(k)$ is defined as the throughput, and $T(k)$ is the average response of the system, $T'$ is an acceptable response time for the user and k is the scalability parameter i.e. number of nodes or users etc. The metric is useful in studying the scalability of distributed systems because the basic objective of scaling up a distributed system is to support more throughput for a fixed response time or to reduce the response time for a fixed throughput or a combination of both [101]. In order to incorporate costs, the authors defined the performance of distributed systems as

$$F(k) = \frac{\lambda(k)}{(1 + \frac{T(k)}{T'}) \times C(k)} \tag{4.2}$$

where C is the cost. Finally, they defined a p-scalability metric $\Psi_p(k_1, k_1')$ that defines the scalability of a distributed system from configuration $k_1$ to $k_1'$ as

$$\Psi_p(k_1, k_1') = \frac{F(k_1')}{F(k_1)} \tag{4.3}$$

In [102], the authors have shown their view of the scalability behavior of a distributed system as illustrated in Figure 4.1. The authors arbitrarily suggested that a distributed system is scalable if

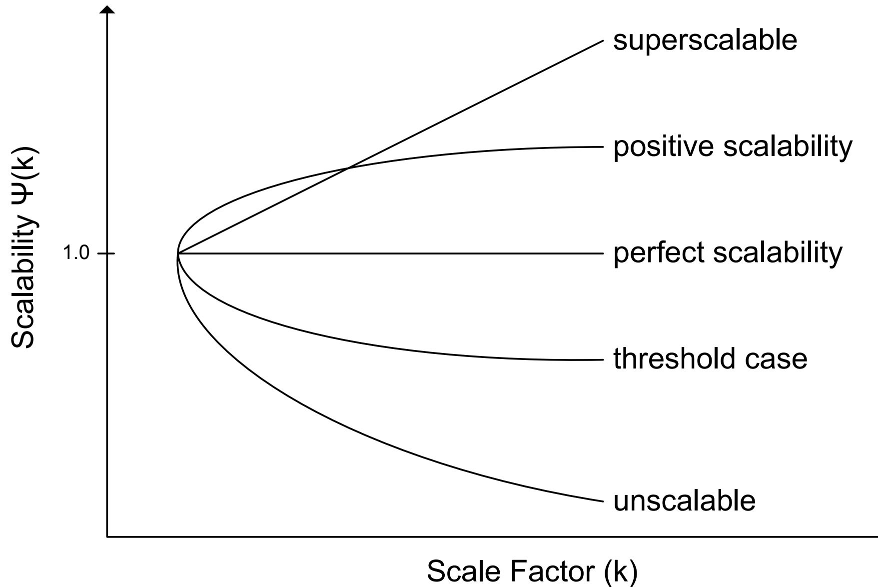



Figure 4.1: The scalability behavior in different situations [102]

$\Psi_p(k_1, k_1') > 0.8$, where 0.8 is a threshold value case to reflect an acceptable benefit-to-cost ratio. However, as might be expected, the focus of this work was on distributed systems and not on distributed algorithms or routing protocols.

To our knowledge, the work presented in [31, 30] is the first preliminary treatment of defining the scalability of an agent-based distributed routing protocol. The authors used equation (4.1) and argued that the throughput and delay of a routing protocol are functions of the number of links (L) and the average hops ($h_{av}$) needed to reach a destination in a particular topology. Therefore,

the power of a routing algorithm, as described in [31, 30] becomes

$$P(k) = \frac{L}{h_{av}} \tag{4.4}$$

Consequently, the power has now become a function of the topology rather than of the algorithm. In their cost model, they simply take the control overhead into consideration. Subsequently, they defined the performance of a routing protocol $F$ as

$$F(k) = \frac{L}{C(k) \times h_{av}} \tag{4.5}$$

where C is the control overhead. Finally, they defined a scalability metric $\Psi(k_1, k_1')$ for a routing protocol as

$$\Psi(k_1, k_1') = \frac{\frac{L_2}{C(k_1') \times h_{av2}}}{\frac{L_1}{C(k_1) \times h_{av1}}} \tag{4.6}$$

where $L_2$, $C(k_1')$ and $h_{av2}$ are number of links, control overhead and average hops, respectively, in topology $k_1'$ and $L_1$, $C(k_1)$ and $h_{av1}$ are number of links, control overhead and average hops, respectively, in topology $k_1$. They took these values from the work of Di Caro and Dorigo [52] and then concluded that *AntNet* is scalable from NSFNet to NTTNet because $\Psi(k_1, k_1') > 1.0$. However, their treatment completely lacked an empirical framework.
In [182], the authors have emphasized the need for an empirical simulation environment that incorporates a real scalable environment for performance evaluation of distributed algorithms. In [107], Katz and Yung have proposed a scalable protocol for authenticated key exchange. They defined four complexity metrics for a network protocol: *round, message, communication and computational.* The *round complexity* is simply the number of rounds until the protocol terminates. The *message complexity* is the total number of messages sent (regardless of their length) by all parties in the course of protocol execution. The *communication complexity* is the total number of bits communicated throughout the execution of the protocol; this now includes the message length as well. *Computation complexity* is the maximum amount of computation done by any player in the protocol. Our cost model is based on some of their ideas as well.
In [45], Costa et al. have proposed a QoS routing algorithm, *Single Mixed Metric (SMM)*, and have done its scalability analysis using a simulation environment. In [165], Santivanez et al. introduced a new metric called *total overhead* for a routing protocol that is defined as follows: *total overhead induced by a routing protocol is the difference between the amount of bandwidth actually consumed by the network running such protocol minus the minimum traffic load that would have been required should the nodes had a priori full topology information* [165]. This metric includes the control overhead, the bandwidth occupied by the control messages, and the suboptimal overhead, resulting from extra hops the packets took in excess of the ideal minimal hop number. However, the focus of this work was to analytically study the asymptotic scalability with respect to a scalable parameter. Nevertheless, the *total overhead* is a valuable concept in analyzing the communication cost of a routing protocol.
In [237], Woodside proposed a scalability metric for analyzing the scalability of mobile agent systems. The work is an extension of his earlier work in scalability of distributed systems reported in [101, 102, 103]. He evaluated the new model on a class of mobile agents, using basic and robust models for the workload and delay. However, he completely ignored the cost in his analysis.

### 4.1.2 Organization of the chapter

We will introduce our scalability model in Section 4.2 and a description of the simulation environments is presented in Section 4.3. Section 4.4 provides a comprehensive description of the results obtained from extensive experiments on a set of topologies varying in their size and complexity. We will then discuss the empirical results obtained from our scalability model in Section 4.5 and then comment on the scalability of the algorithms. Finally, we provide a summary of the chapter.

## 4.2 The scalability model for a routing algorithm

In a sense, a distributed routing algorithm, is a distributed communication system with an objective to optimize throughput and reduce the packet delay. However, the algorithm has to achieve this objective with minimal costs. We could, therefore, define the power of a routing algorithm as a function of a number of performance values, which are collected by our comprehensive performance evaluation framework. In the following subsections we will define the cost model, power model and scalability model of a routing algorithm in a chronological order.

### 4.2.1 Cost model

A distributed routing algorithm is a piece of software that has many associated costs with it. The costs that we will consider in our scalability model are: *processing, communication, and router resources*. We will ignore the costs related to the development, implementation, testing, installation and maintenance of a routing algorithm that arise in a network of real world routers. Instead we will consider the cost of one important resource of a router, which is memory. Now, we will define a few parameters that will influence our cost model:

- *Control processing ratio $C_p$:* is the ratio of the number of cycles that a node spends in processing the control packets, to the number of bits that have been delivered in the network.
- *Data processing ratio $C_d$:* is the ratio of the number of cycles that a node spends in switching the data packets, to the number of bits that have been delivered in the network.
- *Processing ratio $C_\beta$:* is the ratio of the control processing ratio to the data processing ratio i.e $C_\beta = \frac{C_p}{C_d}$.
- *Total overhead $C_t$:* is the sum of the control overhead and suboptimal overhead which have been defined in Chapter 3.
- *Bandwidth ratio $C_w$:* is the ratio of the total extra bits, which are generated due to the total overhead, to the total number of bits that are delivered at their destination.
- *Memory overhead $C_m$:* is the cost associated with storing routing tables in the memory of a router.

$C_p$ and $C_d$ define the processing complexity with respect to the number of bits that are delivered at their destination while $C_\beta$ defines the relative processing overhead of control packets with respect to the packet switching. The motivation for a similar parameter is justified in [116]. A routing algorithm should spend only a fraction of time in processing the control packets as compared to packet switching, which is its actual task [242]. A smaller ratio is definitely desirable for a routing protocol. $C_t$ provides the information about the extra bits, due to total overhead, that have been propagated in the network per time unit in relation to the total available bandwidth in the network. $C_w$ is important at low network traffic loads because it is a function of the number of delivered bits. On the other hand, $C_t$ is important for larger network traffic loads because it increases with an increase in the network traffic. We believe that both of these costs are complementary to capture the communication costs of a routing algorithm.
The amount of memory needed to store the routing tables is directly proportional to the number of entries in the routing tables, therefore, our memory cost model is defined as

$$C_m = \frac{R}{D \times log(\frac{L}{h_{av}})} \tag{4.7}$$

where R is number of entries in the routing table, D total number of nodes in the network, L total number of links and $h_{av}$ is the average number of hops needed to reach a destination. In [31], Carrillo et al. have shown that the number of links between any (source, destination) pair are a function of the total number of links in the network, and the average number of hops ($\frac{L}{h_{av}}$). The

exact combination, however, depends on the routing algorithm and on the topology. Therefore, $\frac{R}{log(\frac{L}{h_{av}})}$ represents the entries needed to model the number of links between a node and other destinations. In this way, the cost associated with larger routing tables is defined as a function of the number of entries in the routing tables and the characteristics of a given topology. So the total cost $C$ in our cost model is

$$C = 1 + C_\beta + e^{C_t} + C_w + C_m \tag{4.8}$$

$e^{C_t}$ is used to emphasize the influence of the total overhead, if $C_t \geq 0.001$, because it is a vital parameter to model the communication cost of a routing algorithm. However, its value otherwise is relatively small by virtue of its definition. We have added a constant cost of 1 to take care of the costs related to the operating system functions like context switching, interrupt processing and network stack processing. We assume that these costs are the same for all routing protocols, which appears reasonable.

### 4.2.2 Power model of an algorithm

In [101, 102, 103], the authors, as discussed in Section 4.1.1, have defined the power of a distributed system as

$$P = \frac{T_{av}}{1 + \omega} \tag{4.9}$$

where $\omega$ corresponds to the ratio of average response time to acceptable response time. In case of a routing protocol $\omega$ becomes the ratio of average packet delay ($t_d$) to the acceptable packet delay ($\frac{t_d}{t_{acc}}$). For our purpose, we defined $t_{acc} = K * t_{min}$ where $t_{min}$ is the minimal packet delay achieved by any of the algorithms at MSIA = 4.6 sec and K is a constant (1.5 in our case) .
We would like to mention that even this new definition of power is unable to capture the influence of important performance values of a routing protocol like jitter, session completion ratio and the standard deviation of the packet delay distributions. Therefore, we now define new parameters that will enable us to have a comprehensive formula for the power of a routing algorithm. The session ratio $\rho$ is defined as

$$\rho = (p_d)^{(2\times(1-S_c))} \tag{4.10}$$

$p_d$ and $S_c$ correspond to packet delivery ratio and session completion ratio respectively, and they are defined in Chapter 3. $\rho$ captures the effect of packet delivery ratio coupled to the session completion ratio. A good routing algorithm should have packet delivery ratio and session completion ratio values as large as possible. With this, we will define our *scaled throughput* $T_{net}$ as a function of this performance value

$$T_{net} = (T_{av})^{1+\frac{\rho}{a}} \tag{4.11}$$

The value of $\rho$ has been scaled down by a constant value "a" (currently 10).
$\zeta$ is the ratio of the 90th percentile of the packet delays to the average delay ($t_{90d}/t_d$). A smaller value indicates that the algorithm has been able to deliver the majority of the packets within small deviation from the average delay. In order to incorporate $\zeta$ in our power formula, we will define $\kappa$

$$\kappa = 1 - e^{-\frac{\omega}{\zeta}} \tag{4.12}$$

This equation gives weight to the value of $\zeta$ only if its value is in proportion to the value of $\omega$. Otherwise if $\omega << \zeta$ then the influence of $\zeta$ is deemphasized. Similarly, the *jitter ratio* $\chi$ is defined as the ratio of the average jitter ($J_d$) to the acceptable jitter $J_{acc}$ ($J_{acc}$ is set to 30 msec). We now define a variable $\sigma$ with the motivation that $\chi$ could only influence the power of a routing algorithm if its value is significantly greater than $J_{acc}$ and comparable with the sum of $\zeta$ and $\omega$.

$$\sigma = 1 - e^{-\frac{\chi}{(\omega+\zeta)}} \tag{4.13}$$

We define a quality-of-service value, $\Upsilon$, in the following

$$\Upsilon = 1 + \omega + \kappa + \sigma \tag{4.14}$$

Now we define the power of a distributed routing algorithm $\gamma$ as

$$\gamma = \frac{T_{net}}{\Upsilon} \tag{4.15}$$

We believe that equation 4.15 models the power of a distributed routing algorithm as a function of a number of relevant performance values. Finally, we define the productivity $\Gamma$ of a routing algorithm as

$$\Gamma = \frac{\gamma}{C} \tag{4.16}$$

### 4.2.3 Scalability metric for a routing algorithm

We here define our scalability metric in the same spirit as described in [102, 103, 31]. First, we define the scalability value $\Omega$ of a distributed algorithm with respect to a number of scalability parameters $k_1, k_2, \ldots k_n$ as

$$\Omega(k_1, k_2, \ldots k_n) = \frac{\Gamma}{k_1 \times k_2 \ldots \times k_n} \tag{4.17}$$

Now we are interested whether the algorithm is scalable at a new value $k'_1$ if compared to a scalable parameter $k_1$. We can reach this decision by defining a scalability metric $\Psi(k_1, k'_1)$ as following

$$\Psi(k_1, k'_1) = \frac{\Omega(k'_1, k_2, \ldots, k_n)}{\Omega(k_1, k_2, \ldots, k_n)} \tag{4.18}$$

We now suggest the following classifications for defining the scalability of a routing protocol. The classification is based on our experience in evaluating the algorithms (*BeeHive*, *AntNet* and *OSPF* on large topologies). We say that a routing algorithm is

- *perfectly scalable* if $\Psi(k_1, k'_1) \geq 1$.
- *positively scalable* if $0.9 \leq \Psi(k_1, k'_1) < 1$.
- *nearly scalable* if $0.8 \leq \Psi(k_1, k'_1) < 0.9$.
- *marginally scalable* if $0.7 \leq \Psi(k_1, k'_1) < 0.8$.
- *not scalable* if $\Psi(k_1, k'_1) < 0.7$.

## 4.3 Simulation environment for scalability analysis

The simulation environment has been already introduced in Chapter 3. We tested the algorithms on six network instances: simpleNet, NTTNet, Node150, Node350, Node650 and Node1050.

### 4.3.1 simpleNet

simpleNet is defined in Chapter 3 and we will refer to this topology with symbol n8 in rest of the chapter.

### 4.3.2 NTTNet

NTTNet is defined in Chapter 3 and we will refer to this topology with symbol n57 in rest of the chapter.

### 4.3.3 Node150

Node150 is defined in Chapter 3 and we will refer to this topology with symbol n150 in rest of the chapter.

### 4.3.4 Node350

Node350 is a 350 nodes network with 464 bidirectional links. The link bandwidth is uniformly distributed between 7 and 14 Mbits/sec, and the propagation delay is uniformly distributed between 1 and 5 msec. The topology was generated using the BRITE software. The Node350 topology is shown in Figure 4.2. We will refer to this topology with symbol n350 in rest of the chapter.

Figure 4.2: Node350: figure is captured from OMNeT++ plotter

### 4.3.5 Node650

Node650 is a 650 nodes network with 785 bidirectional links. The link bandwidth is uniformly distributed between 15 and 20 Mbits/sec, and the propagation delay is uniformly distributed between 1 and 6 msec. The topology was again generated using the BRITE software. The Node650 topology is shown in Figure 4.3. We will refer to this topology with symbol n650 in rest of the chapter.

### 4.3.6 Node1050

Node1050 is a 1050 nodes network with 1295 bidirectional links. The link bandwidth is uniformly distributed between 11 and 20 Mbits/sec, and the propagation delay is uniformly distributed between 1 and 5 msec. The topology was generated using the BRITE software. The Node1050 topology is shown in Figure 4.4. We will refer to this topology with symbol n1050 in rest of the chapter.

## 4.4 Discussion of the results from the experiments

In this section we discuss the behavior of the algorithms based on important performance values, obtained from extensive experiments in OMNeT++, like *throughput, packet delay, packet delivery ratio, session completion ratio, control overhead, suboptimal overhead, agent processing complexity, packet switching complexity and the size of the routing tables.* The objective of this exercise is to study the behavior of the algorithms with respect to the above-mentioned performance values in large topologies. However, we would like to mention that our primary objective is to focus on

Figure 4.3: Node650: figure is captured from OMNeT++ plotter

Figure 4.4: Node1050: figure is captured from OMNeT++ plotter

the scalability behavior of *AntNet* and *BeeHive*, and not on the congestion control behavior as in Chapter 3. The reason for not investigating the congestion control behavior in large topologies is that we do not have the computational resources to simulate congested load scenarios for large topologies. Nevertheless, we have included *OSPF* in our comparison just as a benchmark because it is a state-of-the-art algorithm for normal/static traffic loads. Please note that *OSPF* is a single-path routing algorithm, therefore, it can not scale with an increase in the network traffic load as suggested by the results presented in Chapter 3 and by Di Caro and Dorigo in [52].
The congestion state for large topologies with greater bandwidths could not be reached with the same parameters for the traffic generator as in the smaller topologies. The major emphasis of the work is to answer the question: *can Nature inspired stochastic routing algorithms, like* AntNet *and* BeeHive*, competently perform routing in large topologies?* The answer will be of great significance to the networking community in general and to the Nature inspired routing community in particular.
We gradually decreased the value of MSIA from 4.6 sec to 1.6 sec in all of our experiments. As discussed earlier, MSIA = 2.6 sec or below can cause a network congestion in n57 but for n650 this might be a normal load. This is due to the fact that the links in n57 have a bandwidth of 6 Mbits/sec while in n650 they are on the average of 19 Mbits/sec. The session size was 2130000 bits, the packet size was 512 bytes and MPIA = 0.005 sec. The buffer size for storing data packets in routers was limited to 1000 packets. *BeeHive* and *OSPF* were given 30 seconds to initialize the routing tables. In comparison *AntNet* was given 500 seconds to initialize the routing tables as done by Di Caro and Dorigo in [52]. In all of the reported experiments, the bee generation interval is 1 second, the short distance limit is 7 hops. The performance values are obtained from 1000 seconds of experiments unless otherwise specified. They are an average of the values obtained from ten independent runs for n8 and n57, five independent runs for n150, and three independent runs for n350, n650, n1050. We have to reduce the independent runs due to limited availability of high performance computers. Even then, the burdensome effort of the testing took more than 6 months on our simulation server. Out of 6 months, it took approximately 2 months to evaluate the algorithms on n650 and n1050. Even our current simulation server does not have enough resources to simulate either congested network traffic load in larger topologies, or topologies greater than 1050 nodes. The worst scenario is with n1050 and MSIA = 1.6 sec, in which case we could only simulate 150 seconds of the network traffic, therefore, the results from this scenario are provided to indicate just a tendency. We are actively pursuing our efforts to explore the opportunities to even simulate congested loads in large topologies for 1000 seconds.

### 4.4.1 Throughput and packet delivery ratio

Figure 4.5 and Figure 4.6 show the throughput and the packet delivery ratio of the algorithms as the size of the topology is increased from 8 nodes to 1050 nodes. Each figure consists of three sub-figures which show the behavior of the algorithms at a particular MSIA value. Figure 4.5 shows that all of the three algorithms are able to maintain approximately the same throughput till n150. As the topology grows to 350 nodes or more, the throughput of *AntNet* significantly starts trailing the other algorithms. The reader has to correlate the throughput with the packet delivery ratio in Figure 4.6 to get a comprehensive picture. The larger throughput values in Figure 4.5 might mislead on the conclusion that the behavior of *BeeHive* and *OSPF* are the same. Figure 4.6(a) shows that at MSIA = 4.6 sec, three algorithms are able to deliver approximately all of the packets till n150. Beyond this size topology the packet delivery ratio of *AntNet* dropped from 99% to about 70%, 40%, 30% in n350, n650 and n1050 respectively. *OSPF* is able to deliver all packets except in n57 and n1050. In Figure 4.6(b) we can observe a similar tendency for *AntNet* at MSIA = 2.6 sec as in the previous case. *OSPF* starts significantly trailing *BeeHive* in n57 and n1050. n57 appears to be a complex topology with a low degree of connectivity as compared with other topologies, therefore, the performance of *OSPF* significantly degrades at this instance. The same tendency for the packet delivery ratio of *AntNet* and *OSPF* can be seen in Figure 4.6(c) at MSIA = 1.6 sec, as in the previous case. The packet delivery ratio of *OSPF* is significantly lower

as compared with both *AntNet* and *BeeHive* in n57. However, as might be expected, the packet delivery ratio of *AntNet* significantly degrades after n150. *BeeHive*, at MSIA = 1.6 sec, is able to maintain a significantly higher packet delivery ratio than *OSPF* in all topologies, however, due to scaling problems in Figure 4.5, the same difference is not obvious in the throughput.

We can easily conclude from this series of the experiments that *BeeHive*, as far as throughput is concerned, is able to scale to larger topologies. However, *AntNet*, in comparison, scales well till n150 topology but its throughput significantly deteriorates in n350 or larger topologies. The other performance values of interest are collected in Table 4.1, Table 4.2 and Table 4.3.

| Topology | Algorithm | $t_d$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $P_{drop}$ | $S_c$ | $S_d$ | $S_{90d}$ | $J_d$ | $J_{90d}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | OSPF | 2.99 | 0 | 1 | 1.92 | 0 | 99.7 | 2602 | 2749 | 4 | 9.99 | - | - |
| | AntNet-Co | 2.99 | 0.003 | 0 | 2.18 | 0 | 99.7 | 2601 | 2747 | 4 | 9.99 | 70555 | 6894 |
| | BeeHive | 3.26 | 0 | 0.019 | 2.29 | 0 | 99.7 | 2603 | 2751 | 4.99 | 10.9 | 17920 | 4107 |
| n57 | OSPF | 254 | 0 | 49.6 | 6.8 | 2.51 | 81.4 | 2736 | 3108 | 4 | 9.99 | - | - |
| | AntNet-Co | 72.4 | 2.87 | 5.59 | 8.69 | 0.195 | 96.9 | 2673 | 2933 | 9.29 | 80.8 | 88373 | 14974 |
| | BeeHive | 30.1 | 2.5 | 0.946 | 7.81 | 0.029 | 99.4 | 2629 | 2788 | 7 | 28.6 | 26624 | 11558 |
| n150 | OSPF | 48.8 | 0 | 16.2 | 5.4 | 0.29 | 96.4 | 2638 | 2820 | 4 | 10 | - | - |
| | AntNet-Co | 111 | 2.45 | 12.6 | 8.85 | 0.89 | 96.7 | 2716 | 3395 | 13.4 | 188 | 110235 | 22808 |
| | BeeHive | 22.1 | 0.594 | 0.3 | 5.62 | 0.003 | 99.6 | 2621 | 2769 | 6 | 15 | 34090 | 13648 |
| n350 | OSPF | 31 | 0 | 8.99 | 7.66 | 0.099 | 98.3 | 2626 | 2781 | 4 | 10 | - | - |
| | AntNet-Co | 109 | 10.1 | 11 | 15.5 | 28.1 | 54.5 | 2660 | 3058 | 27.3 | 216 | 289068 | 34610 |
| | BeeHive | 25.4 | 0.257 | 0.17 | 7.94 | 0.014 | 99.5 | 2623 | 2771 | 5.33 | 14.6 | 33547 | 16590 |
| n650 | OSPF | 33 | 0 | 0 | 7.42 | 0.03 | 99.4 | 2630 | 2779 | 4 | 10 | - | - |
| | AntNet-Co | 256 | 10 | 12 | 18.1 | 55.9 | 28.7 | 2752 | 3428 | 58.6 | 486 | - | - |
| | BeeHive | 34.8 | 0.099 | 0.06 | 7.72 | 0.006 | 99.5 | 2633 | 2782 | 5 | 13 | - | - |
| n1050 | OSPF | 116 | 0 | 12 | 7.45 | 9.34 | 70.1 | 2634 | 2787 | 4 | 10 | - | - |
| | AntNet-Co | 320 | 6 | 18.5 | 15.2 | 67.2 | 16.7 | 2830 | 4074 | 30.5 | 329 | - | - |
| | BeeHive | 39.1 | 0.227 | 0.804 | 8.13 | 0.115 | 98.1 | 2640 | 2792 | 7 | 27 | - | - |

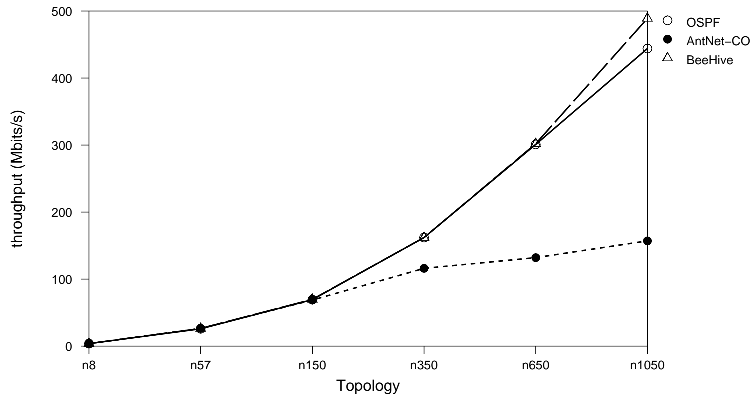
Table 4.1: Performance values for MSIA = 4.6 sec

| Topology | Algorithm | $t_d$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $P_{drop}$ | $S_c$ | $S_d$ | $S_{90d}$ | $J_d$ | $J_{90d}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | OSPF | 2.99 | 0 | 1 | 1.92 | 0 | 99.7 | 2601 | 2747 | 4 | 8.99 | - | - |
| | AntNet-Co | 2.99 | 0.006 | 0 | 2.17 | 0.001 | 99.6 | 2601 | 2747 | 4 | 9.49 | 70893 | 6879 |
| | BeeHive | 3.26 | 0 | 0.02 | 2.27 | 0 | 99.7 | 2602 | 2748 | 4 | 9.99 | 18107 | 4096 |
| n57 | OSPF | 751 | 0 | 149 | 6.21 | 19 | 48.3 | 2677 | 2962 | 4 | 9.89 | - | - |
| | AntNet-Co | 235 | 2.76 | 23.9 | 8.83 | 0.778 | 88.6 | 2825 | 3310 | 26.1 | 182 | 107924 | 15249 |
| | BeeHive | 125 | 3.79 | 12.6 | 8.06 | 0.309 | 93.7 | 2755 | 3143 | 27.9 | 162 | 29733 | 11702 |
| n150 | OSPF | 235 | 0 | 70.6 | 5.34 | 6.75 | 68 | 2663 | 2888 | 4 | 9.19 | - | - |
| | AntNet-Co | 109 | 1.73 | 11.5 | 8.12 | 0.829 | 95.8 | 2710 | 3133 | 14.6 | 146 | 112666 | 23120 |
| | BeeHive | 36.7 | 0.601 | 2.88 | 5.62 | 0.055 | 98.4 | 2642 | 2816 | 7 | 35.8 | 34399 | 13399 |
| n350 | OSPF | 149 | 0 | 36.3 | 7.6 | 4.42 | 73.1 | 2649 | 2836 | 4 | 8.99 | - | - |
| | AntNet-Co | 336 | 9.32 | 32 | 15.5 | 33.8 | 44.9 | 2762 | 3124 | 92.3 | 638 | 292420 | 33429 |
| | BeeHive | 49.2 | 0.29 | 3.13 | 7.96 | 0.29 | 94.3 | 2647 | 2819 | 8 | 44.6 | 36446 | 16296 |
| n650 | OSPF | 73.3 | 0 | 5.33 | 7.41 | 2.04 | 79.7 | 2641 | 2795 | 4 | 8.99 | - | - |
| | AntNet-Co | 316 | 7.82 | 19.6 | 16.3 | 59.9 | 24.6 | 2742 | 3382 | 63 | 492 | - | - |
| | BeeHive | 46.3 | 0.129 | 1.54 | 7.72 | 0.14 | 95.7 | 2645 | 2798 | 6 | 25 | - | - |
| n1050 | OSPF | 160 | 0 | 19 | 7.37 | 21.1 | 54.5 | 2638 | 2796 | 4 | 9 | - | - |
| | AntNet-Co | 391 | 5.29 | 26 | 15.1 | 69.9 | 14.2 | 2766 | 3615 | 49 | 392 | - | - |
| | BeeHive | 80.9 | 0.183 | 5.87 | 8.2 | 1.158 | 77.5 | 2678 | 2866 | 16 | 82 | - | - |

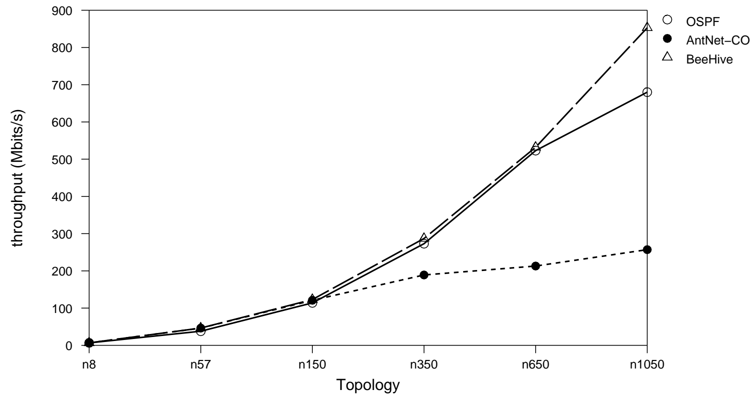
Table 4.2: Performance values for MSIA = 2.6 sec

### 4.4.2 Packet delay

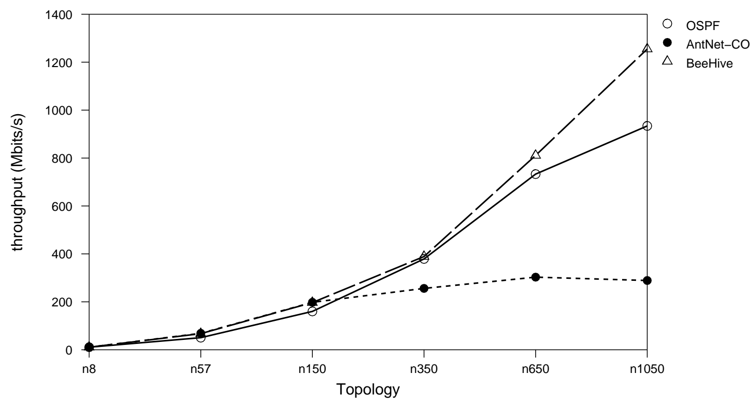
The 90th percentile of the packet delays is another important performance value to study the behavior of a routing algorithm. Figure 4.7 shows the behavior of the packet delay as the size of the topology increases from 8 to 1050 nodes. The values of packet delays are plotted in three different figures for MSIA values of 4.6 sec, 2.6 sec and 1.6 sec respectively. One observation is quite obvious: the n57 topology again appears to be more challenging than the other ones. The




(a) MSIA = 4.6 sec

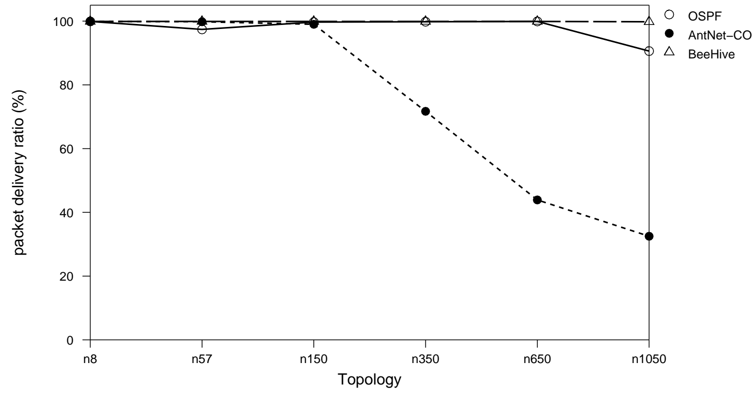



(b) MSIA = 2.6 sec

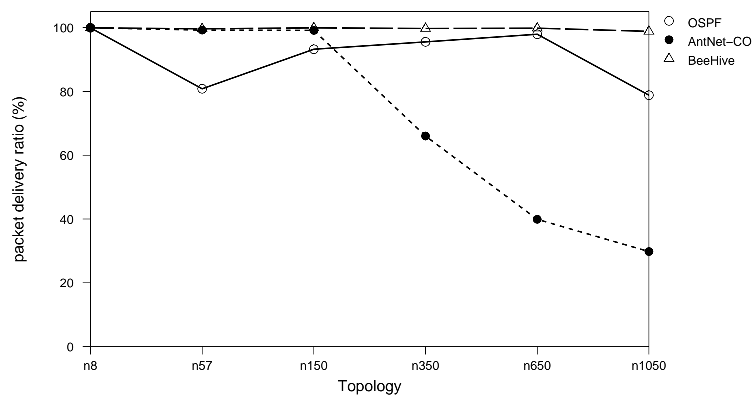



(c) MSIA = 1.6 sec

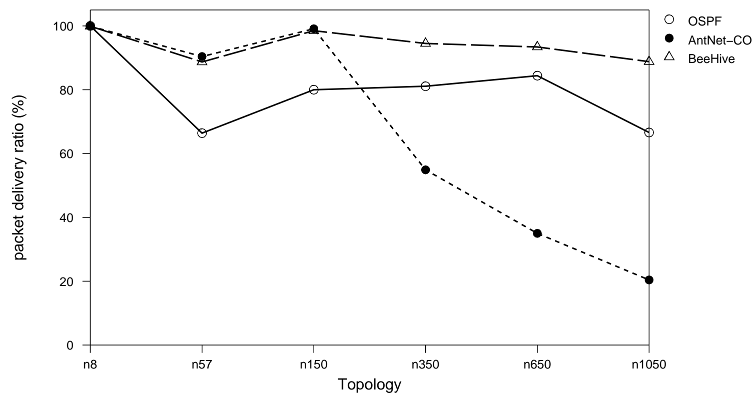
Figure 4.5: Throughput (Mbits/sec))




(a) MSIA = 4.6 sec




(b) MSIA = 2.6 sec




(c) MSIA = 1.6 sec

Figure 4.6: Packet delivery ratio (%)

| Topology | Algorithm | $t_d$ | $P_{loop}$ | $q_{av}$ | $h_{av}$ | $P_{drop}$ | $S_c$ | $S_d$ | $S_{90d}$ | $J_d$ | $J_{90d}$ | $A_a$ | $D_a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | OSPF | 2.99 | 0 | 1 | 1.93 | 0 | 99.7 | 2602 | 2749 | 2.99 | 8 | - | - |
| | AntNet-Co | 2.99 | 0.005 | 0 | 2.18 | 0.001 | 99.6 | 2601 | 2748 | 2.99 | 8 | 71209 | 6852 |
| | BeeHive | 3.3 | 0 | 0.04 | 2.28 | 0 | 99.7 | 2602 | 2749 | 4 | 8.99 | 18086 | 4087 |
| n57 | OSPF | 896 | 0 | 212 | 5.53 | 33.4 | 38.2 | 2695 | 2989 | 2.99 | 8 | - | - |
| | AntNet-Co | 1001 | 3.37 | 112 | 8.77 | 9.44 | 38.5 | 2985 | 3637 | 64.4 | 356 | 138650 | 15824 |
| | BeeHive | 721 | 4.47 | 92.1 | 7.79 | 11.2 | 39 | 2854 | 3391 | 116 | 551 | 108788 | 11742 |
| n150 | OSPF | 384 | 0 | 107 | 5.18 | 19.9 | 49.4 | 2668 | 2891 | 3 | 8 | - | - |
| | AntNet-Co | 223 | 1.66 | 24.8 | 7.9 | 0.813 | 87.3 | 2847 | 3324 | 30 | 221 | 119439 | 23894 |
| | BeeHive | 129 | 0.622 | 19.3 | 5.64 | 1.42 | 80.1 | 2732 | 3016 | 18.2 | 113 | 47451 | 13212 |
| n350 | OSPF | 334 | 0 | 72.3 | 7.41 | 18.8 | 43.2 | 2666 | 2870 | 3 | 8 | - | - |
| | AntNet-Co | 930 | 6.4 | 82 | 13.3 | 44.8 | 8.83 | 2968 | 3491 | 154 | 869 | 280192 | 32586 |
| | BeeHive | 150 | 0.477 | 16.6 | 7.96 | 6.28 | 68.9 | 2679 | 2888 | 18.3 | 109 | 46559 | 16266 |
| n650 | OSPF | 158 | 0 | 17.3 | 7.31 | 15.5 | 48.5 | 2640 | 2795 | 3 | 8 | - | - |
| | AntNet-Co | 425 | 6.26 | 35.9 | 14.9 | 64.7 | 15.2 | 2746 | 3182 | 78.3 | 521 | - | - |
| | BeeHive | 138 | 0.116 | 13.6 | 7.69 | 6.56 | 48.4 | 2654 | 2819 | 14 | 81 | - | - |
| n1050 | OSPF | 215 | 0 | 30 | 7.25 | 33.3 | 38.7 | 2641 | 2802 | 3 | 8 | - | - |
| | AntNet-Co | 579 | 4.8 | 49 | 14.8 | 78.7 | 4.08 | 2785 | 3365 | 117 | 614 | - | - |
| | BeeHive | 197 | 0.084 | 20.6 | 8.19 | 11 | 34.9 | 2676 | 2869 | 29.5 | 157 | - | - |

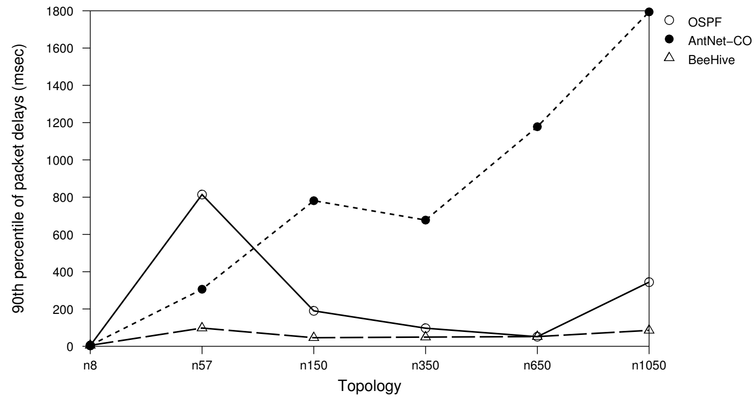
Table 4.3: Performance values for MSIA = 1.6 sec

packet delay of *AntNet*, as shown in Figure 4.7(a), is better than *OSPF* in n57 topology at MSIA = 4.6 sec. The packet delay of *AntNet* then keeps on rising in larger topologies and reaches 1800 msec in n1050, which is significantly higher than that of *OSPF* and *BeeHive*. A similar trend for the packet delay for *AntNet* is observed at MSIA = 2.6 sec and MSIA = 1.6 sec in Figure 4.7(b) and Figure 4.7(c), respectively. The packet delay of *AntNet* becomes significantly greater in n350 and larger topologies. At MSIA = 4.6 sec, *OSPF* has the largest packet delay in n57 and then it gradually decreases to about 400 msec in n1050. Figure 4.7(b) shows that the packet delay of *OSPF* is highest again in n57 and then it gradually drops to 500 msec. A similar trend in packet delay for *OSPF* is observed at MSIA = 1.6 sec. However, at this MSIA value, the difference in the packet delay of *OSPF* compared with other algorithms is relatively smaller. Please note that *BeeHive* has the greatest packet delivery ratio and the smallest packet delay in all of the scenarios.

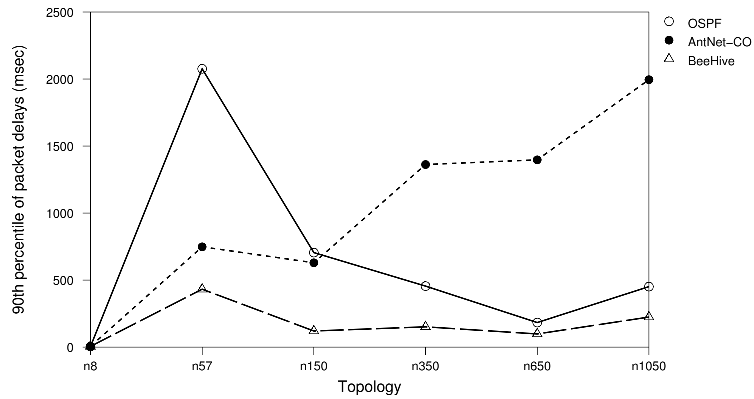
One can find an important conclusion from the experiments: *if the network is in a congestion state, then* BeeHive *is able to do excellent load-balancing as shown by the performance values in n57. Otherwise if the network is not in congestion, then* BeeHive *is able to utilize a shortest path like* OSPF. This adaptive behavior is the result of exploring the multiple paths in parallel by a swarm of replicas of *bee agents* in a small zone around its launching node. Routing the data packets only on those paths that have a quality value above a certain threshold appears to strike a good compromise between maintaining just a single path, as *OSPF* does, and trying to maintain all possible paths as *AntNet* tries to do. Moreover, exploring and managing the paths in a deterministic way, and then distributing the data packets in a stochastic manner on these paths, provides an excellent mix of stochastic elements with deterministic elements. Please note that *OSPF* explores the paths, and routes data packets on them, in a deterministic manner while *AntNet* does both functions in a stochastic manner.

### 4.4.3 Control overhead and suboptimal overhead

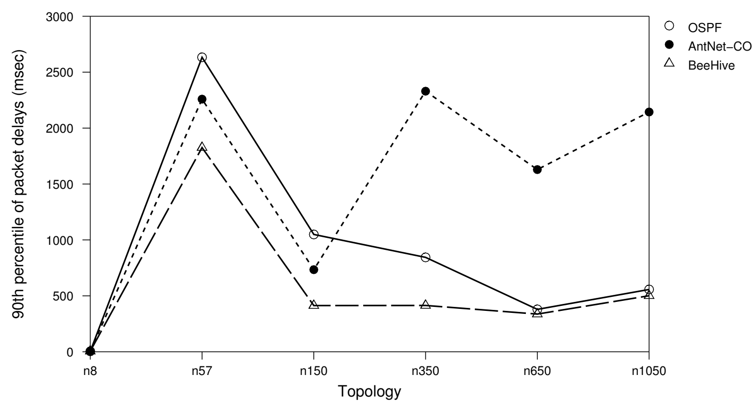
In this subsection we discuss the cost overhead associated with the transmission of the agents and the extra bandwidth consumed by data packets by virtue of taking more hops than in the ideal case. The benefits associated with the stack-less design of *bee agents*, and of gathering the routing information in a local *foraging zone*, are clearly exhibited in Figure 4.8. Please remember that the *bee agents* have a fixed size of 48 bytes. The tendency of control overhead is approximately similar at different values of MSIA: *AntNet* has a overhead comparable to *OSPF* and to *BeeHive* up to n150. Then it sharply increases in larger topologies. The obvious reasons are: first, *ant agents* are equipped with a stack and its size increases with an increase in hops, and second, the stochastic exploration enacts a greater chance to run into loops causing a significant waste of




(a) MSIA = 4.6 sec




(b) MSIA = 2.6 sec




(c) MSIA = 1.6 sec

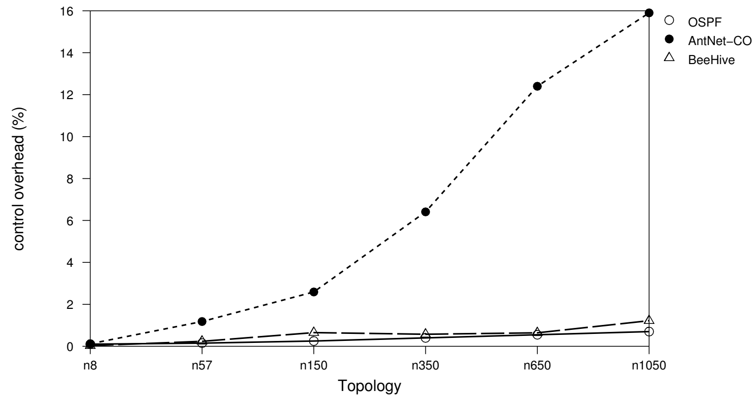
Figure 4.7: Packet delay (msec))

bandwidth. Please note that the control overhead of *BeeHive* at MSIA values of 4.6 sec (see Figure 4.8(a)) and 2.6 sec (see Figure 4.8(b)) is less than or equal to 1% while at MSIA = 1.6 sec (see Figure 4.8(c)) in n1050 it increases to about 2%. However, the control overhead is comparable to *OSPF*. Figure 4.9 shows the suboptimal overhead of the algorithms. Figures 4.9(a), 4.9(b), 4.9(c) show a similar tendency about the suboptimal overhead. Two observations are apparent: one, the suboptimal overhead increases with an increase in the network traffic load, and two, the suboptimal overhead is significantly higher than the control overhead especially in case of n57. One can easily conclude from Figure 4.9 that the suboptimal overhead of *OSPF* is the smallest. This is due to transporting the data packets on shortest paths, however, in congestion scenarios *OSPF* delivers less data packets. The suboptimal overhead of *AntNet* sharply increases after n57 till n350, and then sharply drops at n650 and n1050. The reason for the significant increase is that the data packets which are routed by *AntNet* take more hops ($h_{av}$) to reach their destination, and the number of data packets which follow cyclic paths ($p_{loop}$) also increase significantly in larger topologies. However, the sharp decline in the suboptimal overhead in n650 and n1050 is due to the fact that only 40% of the data packets or less are delivered (see Figure 4.6) at their destination. Please remember that the packets which have been dropped are not included in the suboptimal overhead, therefore, one has to take a comprehensive view by combining Figure 4.9 and Figure 4.6. The suboptimal overhead of *BeeHive* is significantly smaller as compared with *AntNet* in all topologies, and except for n57, it is of the same order as *OSPF*. *BeeHive* is able to maintain a relatively small suboptimal overhead even though its packet delivery ratio is the highest among the three algorithms.

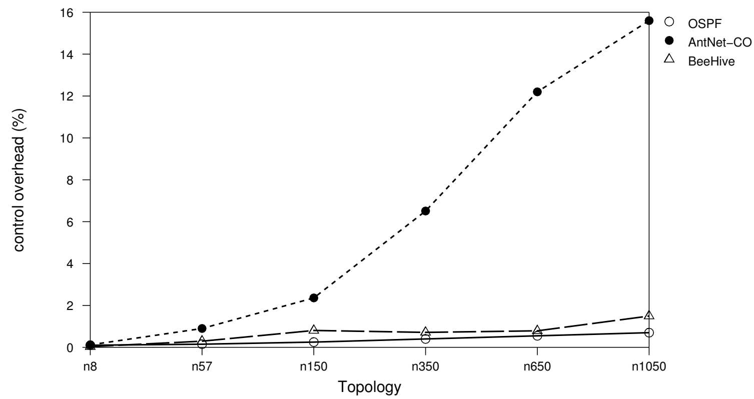
### 4.4.4 Agent and packet processing complexity

The next important performance values are the agent processing complexity and the packet switching complexity of the algorithms. The complexity is measured in terms of the number of cycles the processor needed to process an agent or switch a data packet. We would like to comment that our simulation server could not manage to run the simulations in n650 and n1050 without the need to switch pages from the main memory to virtual memory and vice versa. Therefore, the disk activity interfered with our performance framework, and it effected both agent and data processing complexity parameters. Consequently, we decided not to report these parameters for n650 and n1050 because of the uncertainty caused by the page replacement activity. Nevertheless, the trend obtained from n650 and n1050 was a simple extrapolation of the cycle complexity values in small topologies.

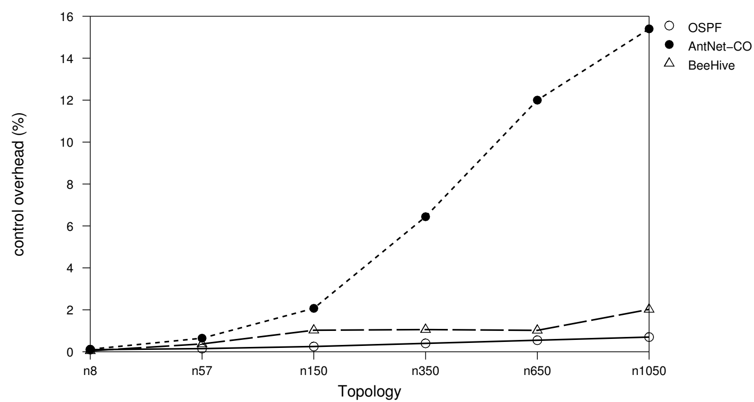
Figure 4.10 shows the number of cycles that a node spends in processing the agents during the experiments. The benefits of stack-less agents, which perform simple mathematical operations and explore the network in a deterministic fashion in a *foraging zone*, are clearly inferable in Figures 4.10(a), 4.10(b), 4.10(c). The general tendency of both *AntNet* and *BeeHive* is the same: an increasing processing complexity with an increase in the size of the network, however, the number of cycles that a node spends in processing *bee agents* are significantly smaller than that for *ant agents*. The difference between the complexity of the *ant agents* and *bee agents* significantly increases in large topologies. Looking at the $A_a$ parameter in Tables 4.1, 4.2, 4.3, it is obvious that the average processing complexity of a *bee agent* does not increase significantly, while in case of an *ant agent* this value increases from 50,000 cycles for n8 to about 290,000 cycles for n350. The packet switching algorithm of *BeeHive* has been carefully designed to have features like no rescaling of the goodness values, and little search time due to smaller routing tables. *BeeHive* only maintains those paths whose quality is above a threshold, therefore, few packets enter into loops. Consequently, all of these factors ensure that a node spends a significantly smaller number of cycles in a packet switching algorithm, which is the main task of an algorithm, in comparison to *AntNet*. Figures 4.11(a), 4.11(b), 4.11(c) show the same tendency: the packet switching complexity of both algorithms is comparable in small topologies like n8 and then the difference between them starts increasing in large topologies like n150 and n350.




(a) MSIA = 4.6 sec

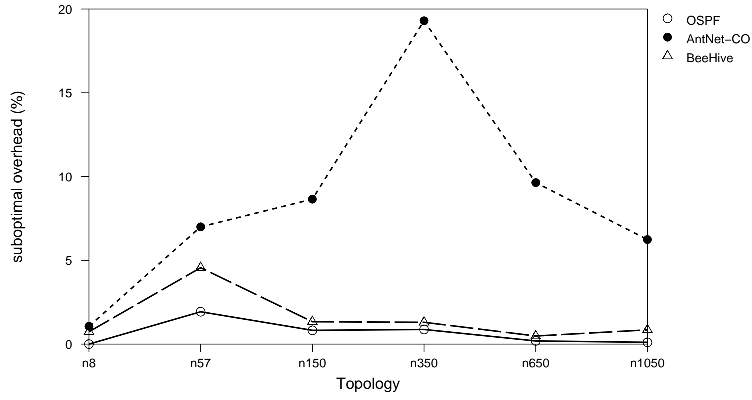



(b) MSIA = 2.6 sec

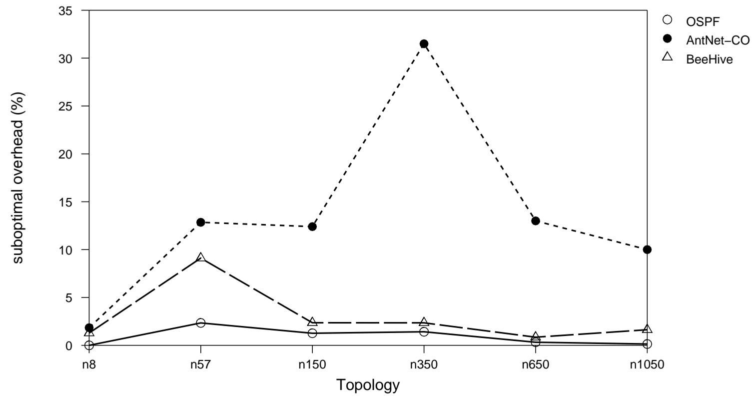



(c) MSIA = 1.6 sec

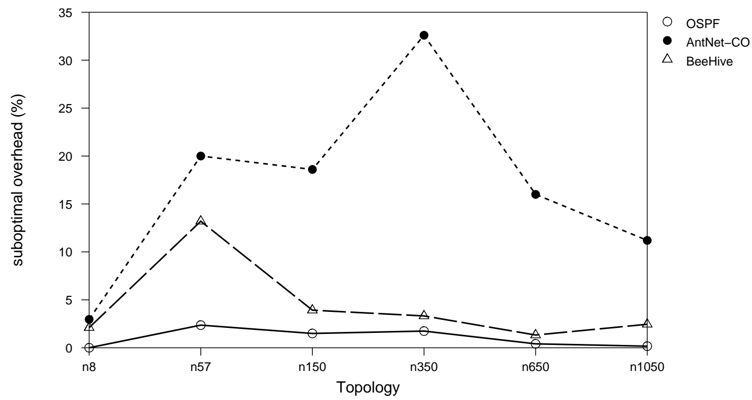
Figure 4.8: Routing overhead (%)




(a) MSIA = 4.6 sec

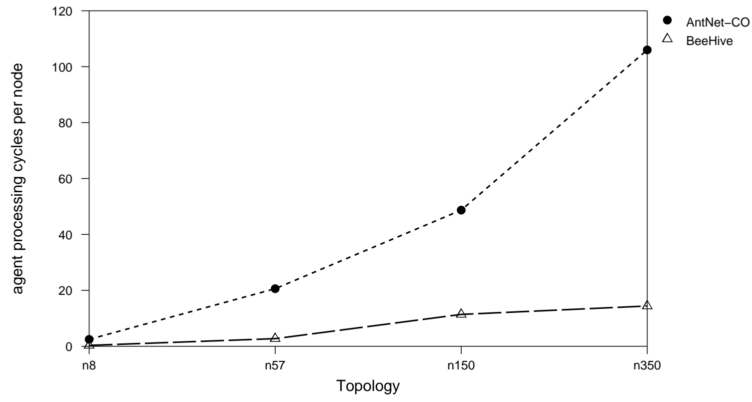



(b) MSIA = 2.6 sec

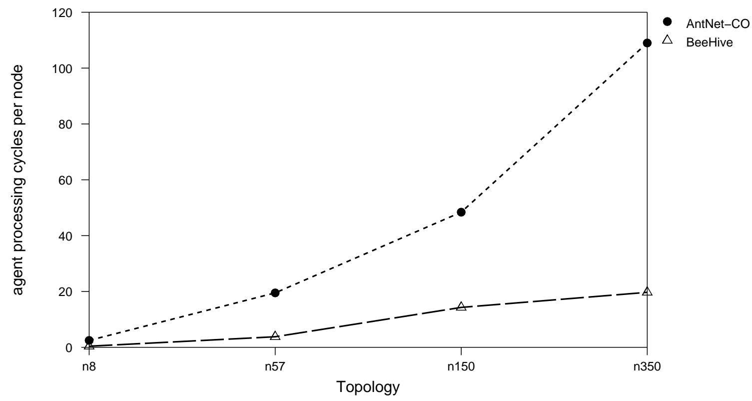



(c) MSIA = 1.6 sec

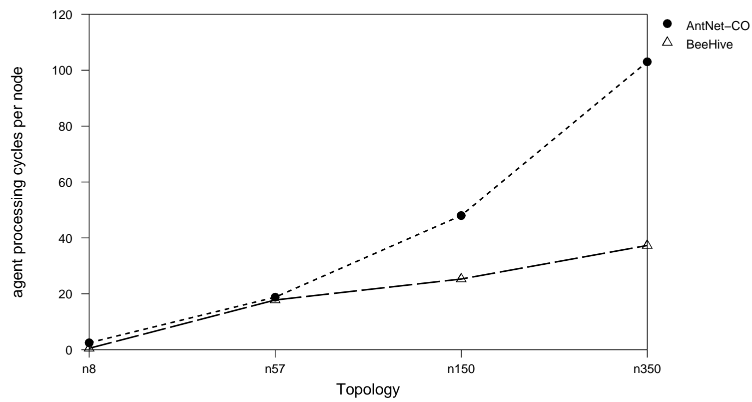
Figure 4.9: Suboptimal overhead (%)




(a) MSIA = 4.6 sec

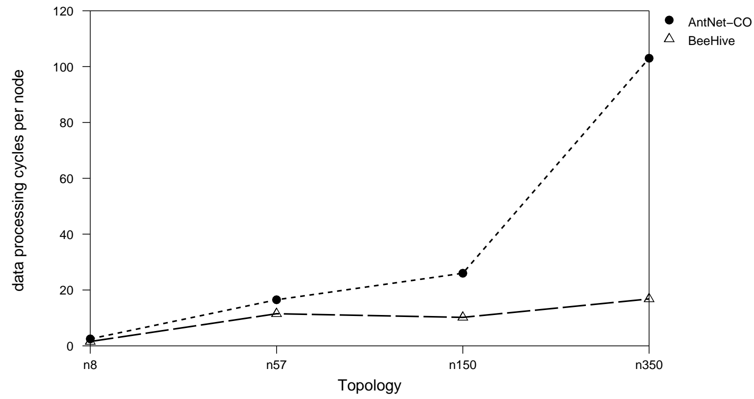



(b) MSIA = 2.6 sec

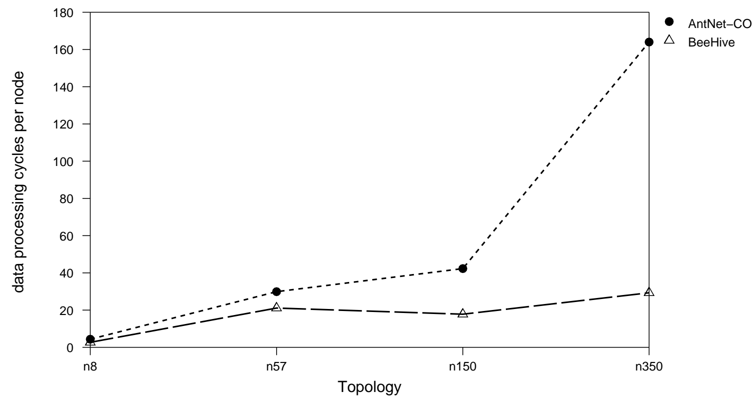



(c) MSIA = 1.6 sec

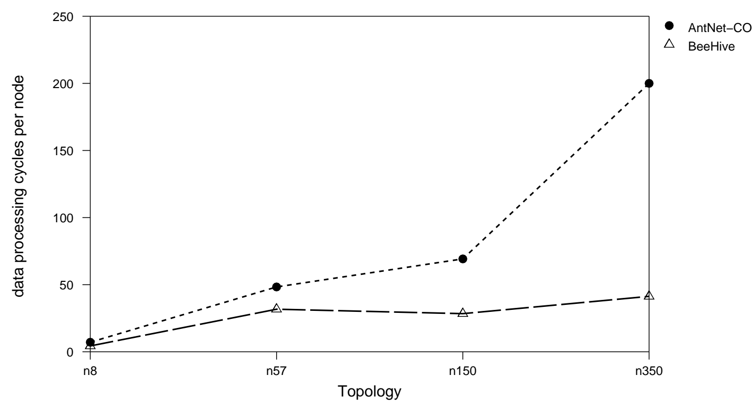
Figure 4.10: Agent processing complexity per node (in billions)




(a) MSIA = 4.6 sec




(b) MSIA = 2.6 sec




(c) MSIA = 1.6 sec

Figure 4.11: Packet switching complexity per node (in billions)

We did not report the processing complexity of *OSPF* because the value depends on how efficient the Dijkstra algorithm is implemented. In order to avoid any controversy, we anyway ignore the processing costs for *OSPF*, i.e $C_\beta = 0$ during the scalability analysis.

### 4.4.5 Routing table size

The benefits of dividing the network into *foraging regions* and *foraging zones* are clearly exhibited in Figure 4.12. The size of the routing table of *BeeHive* is significantly smaller as compared with *AntNet* in larger topologies. Please note that the size of the *BeeHive* routing table includes entries in all three routing tables, and its size is approximately the same as the one for *OSPF*. Running *AntNet* requires, on the average, the same number of entries in the routing tables of a node as the number of the links in the network. Please note that n57, n150, n350, n650, n1050 have 162, 400, 928, 1570 and 2590 bi-directional links, respectively. Figure 4.12 shows clearly that *BeeHive* is able to maintain the best performance in all topologies under all network traffic loads, yet with significantly smaller routing tables.

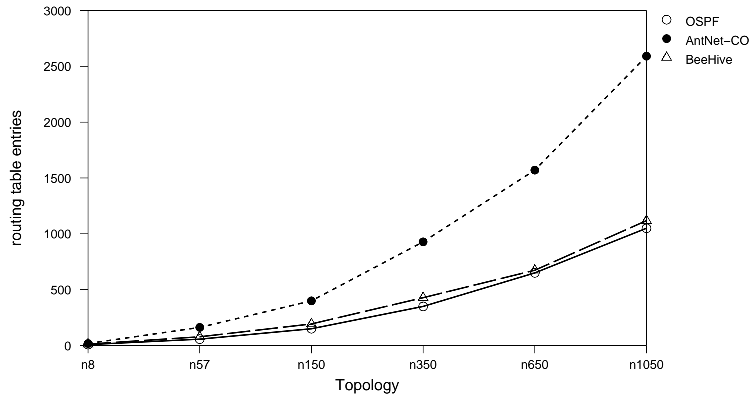



Figure 4.12: The size of the routing table

### 4.4.6 Investigation of the behavior of *AntNet*

The poor performance of *AntNet* in large topologies came as a surprise to us because its performance values are significantly better in n8, n57 and n150. We are in close contact with Di Caro at IDSIA, one of the developers of *AntNet*. He has been kind enough to audit the source code of our implementation of *AntNet*. We decided to investigate the reason for this anomalous behavior, and our performance evaluation framework collected a number of auxiliary parameters that provided us with a valuable insight into the behavior of *AntNet*. After some initial brainstorming we developed two hypotheses for this behavior of *AntNet*:

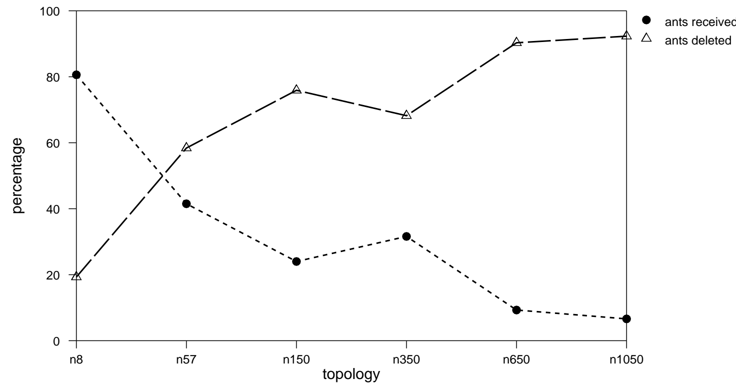
- **H3:** the *ant agents* could not manage the traffic load in large topologies.
- **H4:** the exploration of routes and their maintenance is flawed.

We conducted experiments for MSIA = 10.6 sec and MSIA = 20.6 sec in n350, n650 and n1050, and even at these small network loads the performance of *AntNet* did not improve. Moreover, we noted that if a single path routing algorithm like *OSPF* can manage the load at MSIA = 4.6 sec then a multiple-path routing algorithm like *AntNet* should be able to manage it as well. Subsequently, we focused our investigation on H4 and we collected different parameters from the performance evaluation framework.

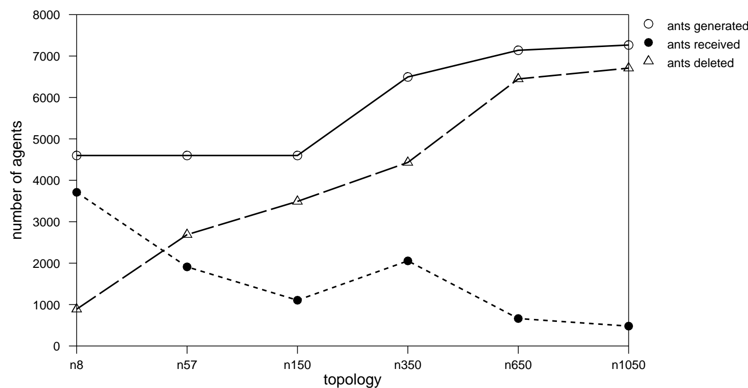
Figure 4.13(a) shows the percentage of *ant agents* that actually managed to return to their source node. It is interesting to note that the percentage of *ant agents* that return to their source node

decreases from 80% in n8 to 40%, 20%, 30%, 10%, 5% in n57, n150, n350, n650 and n1050, respectively. Figure 4.13(b) shows the number of *ant agents* that were launched per node in the network and how many of them actually returned to their source node. Figure 4.13(b) again shows the trend that as the size of the network increases, only a small fraction of the *ant agents* are able to return to their nodes. Moreover, each node in *AntNet*, as suggested by Figure 4.13(b), launches about 4 *ant agents* per second. Even if all of these *ant agents* arrive back at their source node (which of course is not the case), a node in the n650 network would be able to launch an *ant agent* for a destination approximately after every 150 seconds, assuming that all destinations receive the same amount of traffic. The sampling period for a destination increases further to 250 seconds in n1050. This shortcoming coupled with the point-to-point mode of transmission for *ant agents* further degrades the route discovery and management process, especially in large topologies.
Figure 4.14(a) demonstrates the disadvantage of stochastic exploration that allows *ant agents* to loop. The hops values are for the complete journey of an ant, both forward and backward. The average hop count of the received *ant agents* is of the order of the nodes in the network. Consequently, the life time of the *ant agents*, as shown in Figure 4.14(b), sharply increases with an increase in the size of the network. It is obvious that the stochastic network exploration coupled with forward and backward journeys, as expected, is not a promising approach for large networks. We set the TTL value of *ant agents* to $(1.5 \times D)$, therefore, it is highly unlikely that the *ant agents* are dropped because their TTL value has reached zero. Figure 4.14(a) clearly supports this argument as the hop value of the *ant agents* are well below the allowable limit.

Our investigation further revealed that the *ant agents* got deleted because they followed cyclic

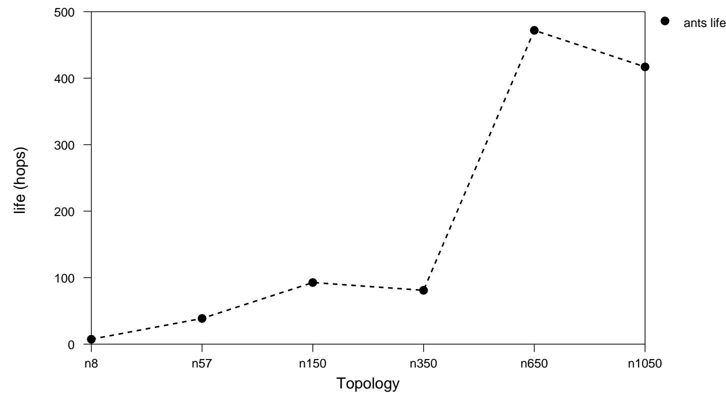



(a) Percentage of ant agents received and deleted

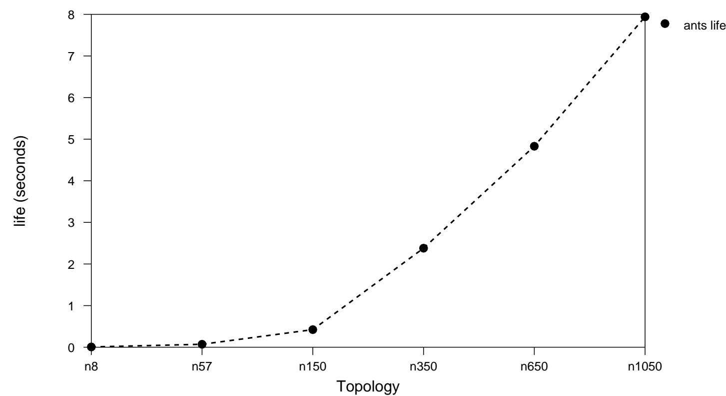



(b) Number of ant agents sent, received and deleted per node

Figure 4.13: The behavior of AntNet (agents sent, received and deleted)




(a) Ants life (hops)




(b) Ants life (sec)

Figure 4.14: The behavior of AntNet (agent life in hops and sec)

paths. Di Caro and Dorigo in [52] kill all those *ant agents* which spend more than half of their life in a cyclic path. The motivation behind this feature in *AntNet* as explained in [52] is: *such an* ant agent *is carrying an old and misleading memory of the network state and it is counterproductive to use it to update the routing tables*. However, our investigation revealed a side effect of this feature: once an *ant agent* follows a cyclic path during the early stage of its network exploration the condition to kill it is nearly always met because it spends more than half of its life in the cycle, as a result, it is killed. A substantial percentage of the *ant agents* that got killed were those which followed a cyclic path by starting at their source node or within few hops of their source node. Consequently, the rate at which the routes leading to different nodes are sampled is significantly reduced. This phenomenon, however, does not yet manifest itself as a serious bottleneck in relatively small topologies like n8, n57 and n150 where still a reasonable number of *ant agents* manage to return to their source nodes, and hence update the routing tables.

## 4.5 Towards empirically founded scalability model for routing protocols

Our comprehensive performance evaluation framework introduced in Chapter 3, proved its usefulness in validating our scalability model because it calculated the performance values utilized by the scalability model. We developed a scalability analyzer tool that retrieves the relevant performance values for calculating the values of different scalability parameters from the database, where the

results of different experiments were stored. Values of relevant parameters are cataloged in Tables 4.4, 4.5, 4.6 for *BeeHive*, *AntNet* and *OSPF* algorithms, respectively. One can summarize from

| Topology | MSIA | $C_p$ | $C_d$ | $C_\beta$ | $C_w$ | $C_t$ | $C_m$ | $C$ | $\zeta$ | $\omega$ | $\kappa$ | $\chi$ | $\sigma$ | $\rho$ | $\Upsilon$ | $T_{net}$ | $\gamma$ | $\Gamma$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | 4.6 | 0.688 | 3.3 | 0.208 | 0.365 | 0.007 | 0.672 | 3.25 | 1.53 | 0.815 | 0.412 | 0.166 | 0.068 | 0.999 | 2.29 | 4.29 | 1.86 | 0.574 |
| | 2.6 | 0.493 | 3.27 | 0.15 | 0.361 | 0.013 | 0.672 | 3.19 | 1.56 | 0.815 | 0.406 | 0.133 | 0.054 | 0.999 | 2.27 | 7.93 | 3.48 | 1.089 |
| | 1.6 | 0.39 | 3.28 | 0.118 | 0.363 | 0.021 | 0.672 | 3.17 | 1.63 | 0.825 | 0.395 | 0.133 | 0.052 | 0.999 | 2.27 | 13.2 | 5.84 | 1.83 |
| n57 | 4.6 | 5.9 | 24.8 | 0.237 | 1.76 | 0.047 | 0.428 | 4.47 | 3.24 | 0.43 | 0.123 | 0.233 | 0.061 | 0.999 | 1.61 | 36.4 | 22.5 | 5.04 |
| | 2.6 | 4.65 | 25.9 | 0.179 | 1.96 | 0.094 | 0.429 | 4.67 | 3.45 | 1.78 | 0.403 | 0.929 | 0.162 | 0.999 | 3.35 | 68 | 20.3 | 4.34 |
| | 1.6 | 15 | 26.8 | 0.561 | 1.96 | 0.135 | 0.426 | 5.09 | 2.53 | 10.3 | 0.982 | 3.86 | 0.26 | 0.863 | 12.5 | 96.8 | 7.71 | 1.51 |
| n150 | 4.6 | 24.5 | 22 | 1.111 | 0.862 | 0.019 | 0.296 | 4.29 | 2.07 | 0.442 | 0.191 | 0.2 | 0.076 | 0.999 | 1.71 | 106 | 62.3 | 14.5 |
| | 2.6 | 17.4 | 21.6 | 0.805 | 0.781 | 0.031 | 0.296 | 3.91 | 3.26 | 0.734 | 0.201 | 0.233 | 0.056 | 0.999 | 1.99 | 199 | 99.9 | 25.5 |
| | 1.6 | 19.2 | 21.5 | 0.892 | 0.762 | 0.049 | 0.296 | 4 | 3.2 | 2.58 | 0.553 | 0.606 | 0.099 | 0.994 | 4.23 | 333 | 78.6 | 19.6 |
| n350 | 4.6 | 31 | 36.2 | 0.857 | 1.222 | 0.018 | 0.241 | 4.34 | 1.94 | 0.508 | 0.23 | 0.177 | 0.069 | 0.999 | 1.8 | 269 | 149 | 34.3 |
| | 2.6 | 23.9 | 35.7 | 0.672 | 1.128 | 0.03 | 0.235 | 4.06 | 3.06 | 0.984 | 0.274 | 0.266 | 0.063 | 0.999 | 2.32 | 505 | 217 | 53.5 |
| | 1.6 | 33.5 | 37.1 | 0.903 | 1.19 | 0.043 | 0.252 | 4.39 | 2.76 | 3 | 0.662 | 0.61 | 0.1 | 0.965 | 4.76 | 691 | 145 | 33 |
| n650 | 4.6 | 0 | 0 | 0 | 1.149 | 0.011 | 0.192 | 3.35 | 1.494 | 0.696 | 0.372 | 0.166 | 0.073 | 0.999 | 2.14 | 534 | 249 | 74.4 |
| | 2.6 | 0 | 0 | 0 | 0.944 | 0.016 | 0.192 | 3.15 | 2.11 | 0.925 | 0.354 | 0.2 | 0.063 | 0.999 | 2.34 | 996 | 425 | 134 |
| | 1.6 | 0 | 0 | 0 | 0.894 | 0.023 | 0.192 | 3.11 | 2.44 | 2.76 | 0.677 | 0.466 | 0.085 | 0.931 | 4.52 | 1514 | 334 | 107 |
| n1050 | 4.6 | 0 | 0 | 0 | 1.76 | 0.02 | 0.181 | 3.96 | 2.18 | 0.782 | 0.3 | 0.233 | 0.075 | 0.999 | 2.15 | 908 | 420 | 106 |
| | 2.6 | 0 | 0 | 0 | 1.53 | 0.031 | 0.181 | 3.75 | 2.76 | 1.61 | 0.442 | 0.533 | 0.114 | 0.994 | 3.17 | 1668 | 525 | 140 |
| | 1.6 | 0 | 0 | 0 | 1.496 | 0.044 | 0.18 | 3.72 | 2.54 | 3.94 | 0.787 | 0.983 | 0.14 | 0.856 | 5.86 | 2312 | 394 | 105 |

Table 4.4: BeeHive

| Topology | MSIA | $C_p$ | $C_d$ | $C_\beta$ | $C_w$ | $C_t$ | $C_m$ | $C$ | $\zeta$ | $\omega$ | $\kappa$ | $\chi$ | $\sigma$ | $\rho$ | $\Upsilon$ | $T_{net}$ | $\gamma$ | $\Gamma$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | 4.6 | 5.3 | 5.35 | 0.99 | 0.285 | 0.011 | 1 | 4.28 | 1.66 | 0.747 | 0.361 | 0.133 | 0.053 | 0.999 | 2.16 | 4.24 | 1.96 | 0.457 |
| | 2.6 | 3.03 | 5.33 | 0.568 | 0.27 | 0.019 | 1 | 3.85 | 1.66 | 0.747 | 0.361 | 0.133 | 0.053 | 0.999 | 2.16 | 7.89 | 3.64 | 0.945 |
| | 1.6 | 1.87 | 5.32 | 0.352 | 0.259 | 0.03 | 1 | 3.64 | 1.66 | 0.747 | 0.361 | 0.099 | 0.04 | 0.999 | 2.14 | 13.4 | 6.24 | 1.71 |
| n57 | 4.6 | 44.5 | 35.7 | 1.246 | 2.99 | 0.163 | 0.883 | 7.29 | 4.22 | 1.034 | 0.217 | 0.309 | 0.057 | 0.999 | 2.3 | 36.6 | 15.8 | 2.17 |
| | 2.6 | 23.9 | 36.8 | 0.651 | 2.88 | 0.275 | 0.883 | 6.73 | 3.18 | 3.35 | 0.651 | 0.87 | 0.124 | 0.998 | 5.13 | 67.8 | 13.2 | 1.96 |
| | 1.6 | 15.6 | 40.1 | 0.388 | 2.91 | 0.412 | 0.883 | 6.7 | 2.25 | 14.3 | 0.998 | 2.14 | 0.121 | 0.883 | 16.4 | 99.6 | 6.06 | 0.905 |
| n150 | 4.6 | 106 | 56.7 | 1.87 | 4.94 | 0.112 | 0.614 | 9.55 | 7.03 | 2.22 | 0.27 | 0.446 | 0.047 | 0.999 | 3.53 | 104 | 29.6 | 3.1 |
| | 2.6 | 59.6 | 52.1 | 1.142 | 3.66 | 0.147 | 0.614 | 7.58 | 5.77 | 2.18 | 0.314 | 0.486 | 0.059 | 0.999 | 3.55 | 195 | 54.9 | 7.24 |
| | 1.6 | 36.3 | 52.3 | 0.694 | 3.17 | 0.206 | 0.614 | 6.71 | 3.29 | 4.46 | 0.742 | 1 | 0.121 | 0.997 | 6.32 | 335 | 53 | 7.9 |
| n350 | 4.6 | 322 | 312 | 1.031 | 11.7 | 0.257 | 0.547 | 15.5 | 6.21 | 2.18 | 0.296 | 0.91 | 0.102 | 0.738 | 3.57 | 164 | 46 | 2.95 |
| | 2.6 | 203 | 305 | 0.664 | 10.6 | 0.38 | 0.547 | 14.3 | 4.05 | 6.72 | 0.809 | 3.07 | 0.248 | 0.632 | 8.77 | 263 | 29.9 | 2.09 |
| | 1.6 | 142 | 274 | 0.517 | 8.09 | 0.39 | 0.547 | 11.6 | 2.5 | 18.6 | 0.999 | 5.13 | 0.215 | 0.335 | 20.8 | 308 | 14.8 | 1.272 |
| n650 | 4.6 | 0 | 0 | 0 | 25.7 | 0.22 | 0.447 | 28.4 | 4.6 | 5.12 | 0.671 | 1.95 | 0.182 | 0.309 | 6.97 | 153 | 22 | 0.774 |
| | 2.6 | 0 | 0 | 0 | 18.3 | 0.252 | 0.447 | 21 | 4.42 | 6.32 | 0.76 | 2.1 | 0.177 | 0.25 | 8.25 | 243 | 29.4 | 1.402 |
| | 1.6 | 0 | 0 | 0 | 14.2 | 0.28 | 0.447 | 17 | 3.83 | 8.5 | 0.891 | 2.61 | 0.19 | 0.168 | 10.5 | 333 | 31.5 | 1.85 |
| n1050 | 4.6 | 0 | 0 | 0 | 29.4 | 0.221 | 0.419 | 32 | 5.6 | 6.4 | 0.68 | 1.016 | 0.081 | 0.153 | 8.16 | 169 | 20.7 | 0.647 |
| | 2.6 | 0 | 0 | 0 | 20.9 | 0.256 | 0.419 | 23.6 | 5.1 | 7.82 | 0.784 | 1.63 | 0.118 | 0.125 | 9.72 | 275 | 28.3 | 1.196 |
| | 1.6 | 0 | 0 | 0 | 19.3 | 0.266 | 0.419 | 22 | 3.7 | 11.5 | 0.956 | 3.9 | 0.225 | 0.047 | 13.7 | 296 | 21.5 | 0.976 |

Table 4.5: AntNet

Table 4.4, when compared with similar Tables 4.5 and 4.6 for *AntNet* and *OSPF*, respectively, that *BeeHive* has comparable power and productivity as *OSPF* and *AntNet* in n8. As expected, *OSPF* has the best power and productivity values in this topology. However, in n57 *OSPF* has the worst power among all algorithms. This is due to its poor performance values: small throughput and significantly high packet delays (please refer to Section 4.4). *BeeHive* has the best power and productivity in n57 at MSIA = 4.6 sec and MSIA = 2.6 sec. At MSIA = 1.6 sec *BeeHive* and *OSPF* have similar productivity values while the productivity of *AntNet* is the smallest. *AntNet* is trailing *BeeHive* because it achieves similar performance as compared with *BeeHive*, but at significantly higher communication, processing and memory costs. Moreover, n57, as mentioned before, is a challenging topology, therefore, congestion can already result at smaller network traffic loads.

n150 appears to be a simple topology with a high degree of connectivity, therefore, the power and productivity of all algorithms are significantly higher in this topology as compared to n57. In n150, *BeeHive* has the best power and productivity for all network traffic loads except at MSIA = 4.6

| Topology | MSIA | $C_p$ | $C_d$ | $C_\beta$ | $C_w$ | $C_t$ | $C_m$ | $C$ | $\zeta$ | $\omega$ | $\kappa$ | $\chi$ | $\sigma$ | $\rho$ | $\Upsilon$ | $T_{net}$ | $\gamma$ | $\Gamma$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n8 | 4.6 | 0 | 0 | 0 | 0.048 | 0.001 | 0.444 | 2.49 | 1.337 | 0.747 | 0.428 | 0.133 | 0.061 | 0.999 | 2.23 | 4.24 | 1.89 | 0.76 |
| | 2.6 | 0 | 0 | 0 | 0.027 | 0.001 | 0.444 | 2.47 | 1.337 | 0.747 | 0.428 | 0.133 | 0.061 | 0.999 | 2.23 | 7.95 | 3.55 | 1.438 |
| | 1.6 | 0 | 0 | 0 | 0.016 | 0.001 | 0.444 | 2.46 | 1.337 | 0.747 | 0.428 | 0.099 | 0.046 | 1 | 2.22 | 13.4 | 6.04 | 2.45 |
| n57 | 4.6 | 0 | 0 | 0 | 0.781 | 0.02 | 0.31 | 3.11 | 3.2 | 3.62 | 0.677 | 0.133 | 0.019 | 0.99 | 5.32 | 35.5 | 6.68 | 2.14 |
| | 2.6 | 0 | 0 | 0 | 0.636 | 0.024 | 0.31 | 2.97 | 2.76 | 10.7 | 0.979 | 0.133 | 0.009 | 0.802 | 12.7 | 50.7 | 3.98 | 1.341 |
| | 1.6 | 0 | 0 | 0 | 0.482 | 0.025 | 0.31 | 2.81 | 2.93 | 12.8 | 0.987 | 0.099 | 0.006 | 0.602 | 14.7 | 63.8 | 4.31 | 1.53 |
| n150 | 4.6 | 0 | 0 | 0 | 0.469 | 0.01 | 0.23 | 2.71 | 3.89 | 0.976 | 0.221 | 0.133 | 0.027 | 0.999 | 2.22 | 105 | 47.5 | 17.5 |
| | 2.6 | 0 | 0 | 0 | 0.4 | 0.015 | 0.23 | 2.64 | 3 | 4.7 | 0.791 | 0.133 | 0.017 | 0.955 | 6.5 | 179 | 27.5 | 10.4 |
| | 1.6 | 0 | 0 | 0 | 0.33 | 0.017 | 0.23 | 2.57 | 2.73 | 7.68 | 0.939 | 0.1 | 0.009 | 0.797 | 9.62 | 239 | 24.9 | 9.65 |
| n350 | 4.6 | 0 | 0 | 0 | 0.83 | 0.012 | 0.206 | 3.04 | 3.12 | 0.62 | 0.179 | 0.133 | 0.034 | 0.999 | 1.83 | 269 | 146 | 48.1 |
| | 2.6 | 0 | 0 | 0 | 0.703 | 0.018 | 0.206 | 2.92 | 3.05 | 2.98 | 0.623 | 0.133 | 0.021 | 0.975 | 4.62 | 471 | 102 | 34.8 |
| | 1.6 | 0 | 0 | 0 | 0.598 | 0.021 | 0.206 | 2.82 | 2.52 | 6.68 | 0.928 | 0.1 | 0.01 | 0.788 | 8.61 | 605 | 70.2 | 24.8 |
| n650 | 4.6 | 0 | 0 | 0 | 0.755 | 0.007 | 0.185 | 2.94 | 1.55 | 0.66 | 0.345 | 0.133 | 0.058 | 0.999 | 2.06 | 532 | 258 | 87.5 |
| | 2.6 | 0 | 0 | 0 | 0.514 | 0.008 | 0.185 | 2.7 | 2.49 | 1.466 | 0.444 | 0.133 | 0.033 | 0.991 | 2.94 | 972 | 330 | 122 |
| | 1.6 | 0 | 0 | 0 | 0.403 | 0.009 | 0.185 | 2.59 | 2.39 | 3.16 | 0.732 | 0.1 | 0.017 | 0.839 | 4.9 | 1275 | 259 | 99.9 |
| n1050 | 4.6 | 0 | 0 | 0 | 0.762 | 0.008 | 0.169 | 2.94 | 2.96 | 2.31 | 0.542 | 0.133 | 0.024 | 0.942 | 3.88 | 788 | 202 | 69 |
| | 2.6 | 0 | 0 | 0 | 0.515 | 0.008 | 0.169 | 2.69 | 2.81 | 3.2 | 0.678 | 0.133 | 0.021 | 0.805 | 4.9 | 1149 | 234 | 87 |
| | 1.6 | 0 | 0 | 0 | 0.388 | 0.008 | 0.169 | 2.56 | 2.59 | 4.3 | 0.809 | 0.1 | 0.014 | 0.607 | 6.12 | 1415 | 231 | 90 |

Table 4.6: OSPF

sec, where *OSPF* has the best productivity. However, as might be expected from the experiments reported in Section 4.4 that the power of *OSPF* starts trailing as compared to *AntNet* at MSIA = 2.6 sec, and at MSIA = 1.6 sec under high loads. Yet, it achieves a higher productivity than *AntNet* because of its better benefit-to-cost ratio.

In the n350 topology, *OSPF* has the best productivity among all algorithms at MSIA = 4.6 sec because the network traffic load at this value can not cause a congestion for this instance of network, therefore, it achieves a similar power as *BeeHive*, but with lesser costs. Consequently, its productivity is significantly higher than that of *BeeHive*. However, as the network traffic load is increased by taking the MSIA values of 2.6 sec and 1.6 sec, the power and productivity of *OSPF* significantly starts trailing *BeeHive*. Please note that the power and productivity of *AntNet* in n350 have significantly degraded and are the lowest among all algorithms. This behavior is due to its poor performance as described in Section 4.4.

The power and productivity of the algorithms show the same tendency in the n650 topology. The power and productivity of *BeeHive* is significantly greater among all algorithms at MSIA = 2.6 sec and MSIA = 1.6 sec, however, at MSIA = 4.6 sec values are slightly smaller than *OSPF*. The power and productivity of *AntNet* is further decreased due to its poor performance. Finally, *BeeHive* has again the best power and performance among all algorithms, for all traffic patterns in n1050. Both values are significantly higher than those of *OSPF*. Since n1050 is a relatively difficult topology with a rather low degree of connectivity, the power and productivity of *AntNet* is further deteriorated.

One has to be careful in analyzing the scalability of a routing algorithm based on the scalability metric $\Psi$ introduced in Section 4.2. One drawback of $\Psi$ is that it simply takes the ratio of the productivity values in two configurations and then the ratio of the scalability parameters with respect to which the scalability has to be studied. The ratio factors out the influence of the productivity value itself e.g. if an algorithm A has productivity values of 0.9 and 1.2 in two configurations and another algorithm B has productivity values of 10 and 15 then both algorithms are termed as perfectly scalable although the algorithm A has smaller productivity values in both states. Therefore, one has to always complete the picture by looking both at $\Psi$ and $\Gamma$.

### 4.5.1 Scalability matrix and scalability analysis

In order to comprehensively analyze the scalability of the algorithms, we have developed a *scalability matrix* for each algorithm. The scalability matrix is populated under the following rules:

- The topologies are listed in the first column and the first row. Within each topology, different traffic patterns, modeled by changing the values of MSIA, are listed.

- In this matrix, we are interested in studying the scalability by either keeping the same topology and changing the traffic patterns, or by keeping the same traffic patterns but changing the topology.
- An initial configuration is defined by selecting a topology in the first column followed by choosing an MSIA value. A final configuration is defined by selecting a topology from the first row and then choosing the MSIA value.
- A transition from an initial configuration to a final configuration is only allowed in the matrix if the following three conditions are met:
  - The topology of the final configuration is extending the current configuration (in terms of nodes and/or links).
  - The MSIA value of the final configuration is smaller or equal to the one in current configuration.
  - The MSIA value of the final configuration is equal to the initial configuration if the topologies are different.
- If a transition is allowed, then the corresponding value is simply a ratio of $\frac{\Omega_2}{\Omega_1}$, where $\Omega_2$ and $\Omega_1$ are scalability values in the final and initial configurations, respectively (they are defined in Section 4.2). Basically, we are interested in studying how the algorithm scales for the same traffic patterns once new nodes are added, or how the algorithm scales to additional injection of traffic load but for a fixed topology.
- An invalid transition is marked by a "$\times$".

Let us take the example of the scalability matrix of *BeeHive* illustrated in Table 4.7. If we want to study the scalability of *BeeHive* from n8, MSIA = 4.6 sec to n350, MSIA = 4.6 sec then the corresponding entry is 1.38 which shows that the algorithm is perfectly scalable for these configurations. However, the scalability metric from n57, MSIA = 4.6 sec to n57, MSIA = 2.6 sec is 0.48, which is an indication that the algorithm is not scalable for these configurations. Similarly, the following transitions, for example, are not allowed: n8, MSIA = 4.6 sec to n57, MSIA = 2.6 sec and n57, MSIA = 1.6 sec to n57, MSIA = 2.6 sec. We now discuss the scalability of each algorithm separately.

### 4.5.2 Scalability analysis of BeeHive

One can easily conclude from Table 4.7 that *BeeHive* is perfectly scalable at MSIA = 4.6 sec if n8 is considered as the reference topology. *BeeHive* is also perfectly scalable at MSIA = 2.6 sec except in n57 where it is not scalable and in n1050, where it is positively scalable. However, at MSIA = 1.6 sec, the algorithm is not scalable except for n650 where it is marginally scalable. The algorithm is also perfectly scalable to an additional network traffic load in n8 topology.
The shortcoming of the scalability metric, as discussed before, manifests itself if one takes n57 as the reference. *BeeHive* appears to be perfectly scalable for all network traffic loads including MSIA = 1.6 sec which is counterintuitive. The reason is obvious: the productivity of *BeeHive* is significantly smaller in n57 at MSIA = 1.6 sec, therefore, the scalability metric significantly increases.
If we take n150 as the reference topology then the *BeeHive* algorithm is perfectly scalable at MSIA = 4.6 sec. In n150, the algorithm is nearly scalable, perfectly scalable and marginally scalable for n350, n650 and n1050, respectively, at MSIA = 2.6 sec. A similar tendency is seen at MSIA = 1.6 sec. The algorithm is able to perfectly scale to the additional traffic injected at MSIA = 2.6 sec but not at MSIA = 1.6.
The algorithm is nearly scalable or better, if n350 is taken as the reference topology, for all configurations except one, in which additional load is injected in n350 at MSIA = 1.6 sec. The algorithm also does not scale at MSIA = 2.6 sec and MSIA = 1.6 sec, if n650 is taken as the

reference topology. However, at MSIA = 4.6 sec it is nearly scalable. In n650, the algorithm is perfectly scalable to an additional injected load at MSIA = 2.6 sec, however, it is not scalable at MSIA = 1.6 sec. In n1050 the algorithm does not scale to the additional traffic load at MSIA = 1.6 sec.
*BeeHive is nearly scalable or better in most of the cases with MSIA = 4.6 sec and MSIA = 2.6 sec, however, for MSIA = 1.6 sec it is not scalable.*

| | | n8 | | | n57 | | | n150 | | | n350 | | | n650 | | | n1050 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 |
| n8 | 4.6 | 1 | 1.087 | 1.139 | 1.254 | x | x | 1.365 | x | x | 1.387 | x | x | 1.61 | x | x | 1.42 | x | x |
| | 2.6 | x | 1 | 1.048 | x | 0.563 | x | x | 1.249 | x | x | 1.119 | x | x | 1.52 | x | x | 0.978 | x |
| | 1.6 | x | x | 1 | x | x | 0.114 | x | x | 0.565 | x | x | 0.458 | x | x | 0.713 | x | x | 0.43 |
| n57 | 4.6 | x | x | x | 1 | 0.488 | 0.104 | 1.088 | x | x | 1.106 | x | x | 1.288 | x | x | 1.131 | x | x |
| | 2.6 | x | x | x | x | 1 | 0.213 | x | 2.21 | x | x | 1.98 | x | x | 2.7 | x | x | 1.73 | x |
| | 1.6 | x | x | x | x | x | 1 | x | x | 4.92 | x | x | 3.98 | x | x | 6.21 | x | x | 3.74 |
| n150 | 4.6 | x | x | x | x | x | x | 1 | 0.994 | 0.471 | 1.016 | x | x | 1.183 | x | x | 1.039 | x | x |
| | 2.6 | x | x | x | x | x | x | x | 1 | 0.474 | x | 0.896 | x | x | 1.221 | x | x | 0.783 | x |
| | 1.6 | x | x | x | x | x | x | x | x | 1 | x | x | 0.81 | x | x | 1.261 | x | x | 0.761 |
| n350 | 4.6 | x | x | x | x | x | x | x | x | x | 1 | 0.877 | 0.376 | 1.165 | x | x | 1.023 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | 1 | 0.429 | x | 1.362 | x | x | 0.874 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 1.55 | x | x | 0.939 |
| n650 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 1.025 | 0.503 | 0.878 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.49 | x | 0.641 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 0.603 |
| n1050 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.749 | 0.345 |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.461 |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 |

Table 4.7: Scalability Matrix for BeeHive

### 4.5.3 Scalability analysis of AntNet

The power of *AntNet* has significantly deteriorated in n350 and onwards, therefore, the scalability metric is of little significance. However, in n1050 the algorithm appears to perfectly scale to the additional injected traffic load at MSIA = 2.6 sec because the corresponding $\Psi$ value is 1.038. However, this is due to the productivity values of 0.647 and 1.196 at MSIA = 4.6 sec and 2.6 sec, respectively (see Table 4.5). Consequently, their ratio leads to a value of 1.038. One can easily conclude from the scalability metrics of *AntNet*, as shown in Table 4.8, that the algorithm is not scalable for most of the configurations.

### 4.5.4 Scalability analysis of OSPF

The scalability matrix for *OSPF* is illustrated in Table 4.9. As expected, the general trend is that most of the times *OSPF* is not scalable for MSIA = 1.6 sec, either across different topologies or within the same topology. Moreover, the scalability metrics are generally inferior to *BeeHive*. The exception to this generalization are the cases where the productivity of *OSPF* is extremely low in one topology and then it significantly improves in an another topology. One such anomaly is observed at MSIA = 2.6 sec and n57 as the reference topology. One might conclude that in this case *OSPF* is perfectly scalable and its scalability metrics are also of higher value compared to *BeeHive*. This is due to the poor productivity of *OSPF* in n57, as cataloged in Table 4.6. The productivity of *OSPF* at MSIA = 2.6 sec increases from 1.34 in n57 to 10.4, 34.8, 122 and 87 in n150, n350, n650 and n1050, respectively. In comparison the productivity of *BeeHive* (see Table 4.4) increases from 4.34 in n57 to 25.5, 53.5, 134 and 140 in n150, n350, n650 and n1050, respectively. The complete picture is that *BeeHive* has a higher productivity than *OSPF* in all cases but its scalability metric is less than the one of *OSPF* because its productivity in n57 is significantly higher as compared to *OSPF*. Right now we are pursuing different options to take care of such anomalous behavior in our definition of the scalability metric.

| | | n8 | | | n57 | | | n150 | | | n350 | | | n650 | | | n1050 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 |
| n8 | 4.6 | 1 | 1.177 | 1.309 | 0.668 | x | x | 0.364 | x | x | 0.148 | x | x | 0.02 | x | x | 0.01 | x | x |
| | 2.6 | x | 1 | 1.112 | x | 0.291 | x | x | 0.408 | x | x | 0.05 | x | x | 0.018 | x | x | 0.009 | x |
| | 1.6 | x | x | 1 | x | x | 0.074 | x | x | 0.246 | x | x | 0.016 | x | x | 0.013 | x | x | 0.004 |
| n57 | 4.6 | x | x | x | 1 | 0.513 | 0.145 | 0.544 | x | x | 0.222 | x | x | 0.031 | x | x | 0.016 | x | x |
| | 2.6 | x | x | x | x | 1 | 0.283 | x | 1.402 | x | x | 0.173 | x | x | 0.062 | x | x | 0.032 | x |
| | 1.6 | x | x | x | x | x | 1 | x | x | 3.31 | x | x | 0.228 | x | x | 0.179 | x | x | 0.057 |
| n150 | 4.6 | x | x | x | x | x | x | 1 | 1.32 | 0.885 | 0.407 | x | x | 0.057 | x | x | 0.029 | x | x |
| | 2.6 | x | x | x | x | x | x | x | 1 | 0.67 | x | 0.123 | x | x | 0.044 | x | x | 0.023 | x |
| | 1.6 | x | x | x | x | x | x | x | x | 1 | x | x | 0.069 | x | x | 0.054 | x | x | 0.017 |
| n350 | 4.6 | x | x | x | x | x | x | x | x | x | 1 | 0.401 | 0.149 | 0.14 | x | x | 0.073 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | 1 | 0.373 | x | 0.359 | x | x | 0.189 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 0.783 | x | x | 0.253 |
| n650 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 1.024 | 0.833 | 0.52 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.813 | x | 0.527 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 0.323 |
| n1050 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 1.038 | 0.517 |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.498 |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 |

Table 4.8: Scalability Matrix for AntNet

For now, we propose as a **Principle**: *a reference topology for analyzing the comparative scalability behavior of routing protocols must be chosen in such a fashion that all candidate algorithms should have similar productivity values in it.* This principle is crucial to follow in the current scalability framework because it takes care of the complex relationship between the performance of a routing protocol and the topology. Luckily, n8 is one such topology, therefore, the scalability analysis based on the performance values in this topology is completely unbiased because all of the three algorithms have similar productivity values for all traffic patterns (see Tables 4.4, 4.5, 4.6).

| | | n8 | | | n57 | | | n150 | | | n350 | | | n650 | | | n1050 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 | 4.6 | 2.6 | 1.6 |
| n8 | 4.6 | 1 | 1.07 | 1.127 | 0.397 | x | x | 1.237 | x | x | 1.451 | x | x | 1.423 | x | x | 0.69 | x | x |
| | 2.6 | x | 1 | 1.053 | x | 0.131 | x | x | 0.387 | x | x | 0.557 | x | x | 1.046 | x | x | 0.462 | x |
| | 1.6 | x | x | 1 | x | x | 0.087 | x | x | 0.21 | x | x | 0.231 | x | x | 0.501 | x | x | 0.279 |
| n57 | 4.6 | x | x | x | 1 | 0.353 | 0.249 | 3.11 | x | x | 3.65 | x | x | 3.58 | x | x | 1.73 | x | x |
| | 2.6 | x | x | x | x | 1 | 0.705 | x | 2.95 | x | x | 4.25 | x | x | 7.98 | x | x | 3.52 | x |
| | 1.6 | x | x | x | x | x | 1 | x | x | 2.39 | x | x | 2.63 | x | x | 5.7 | x | x | 3.18 |
| n150 | 4.6 | x | x | x | x | x | x | 1 | 0.334 | 0.191 | 1.172 | x | x | 1.15 | x | x | 0.557 | x | x |
| | 2.6 | x | x | x | x | x | x | x | 1 | 0.572 | x | 1.44 | x | x | 2.7 | x | x | 1.193 | x |
| | 1.6 | x | x | x | x | x | x | x | x | 1 | x | x | 1.099 | x | x | 2.38 | x | x | 1.328 |
| n350 | 4.6 | x | x | x | x | x | x | x | x | x | 1 | 0.411 | 0.179 | 0.981 | x | x | 0.475 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | 1 | 0.436 | x | 1.87 | x | x | 0.828 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 2.16 | x | x | 1.208 |
| n650 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.786 | 0.396 | 0.484 | x | x |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.504 | x | 0.441 | x |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | x | x | 0.557 |
| n1050 | 4.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.716 | 0.456 |
| | 2.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 0.636 |
| | 1.6 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 |

Table 4.9: Scalability Matrix for OSPF

## 4.6 Summary

We have empirically evaluated the behavior of two state-of-the-art Nature inspired routing algorithms and a single-path routing algorithm on a set of topologies ranging from 8 to 1050 nodes. The results unequivocally demonstrate that *BeeHive* is able to deliver superior performance both under high or low network traffic loads, in all topologies. This is a result of considering scalability

as an important factor in the design and development of our *bee agent* model. The true benefits of the algorithm are clearly visible in larger topologies. The performance of *AntNet*, however, significantly degrades in 350 or higher nodes topologies.
We also proposed power and productivity metrics for a distributed routing algorithm, which depend on a number of important performance values. The productivity metric provides an unbiased insight into the behavior of a routing algorithm: at what costs does the algorithm improve its performance? Finally, we defined a scalability metric to reach a decision on the scalability of a routing algorithm and demonstrated that the *BeeHive algorithm scales better than the rest of the algorithms in majority of the cases because of its superior benefit-to-cost ratio.*
The scalability metric suffers from a shortcoming: it takes the ratio of the productivities in two configurations, as a result, it simply factors out the magnitude of the productivity. Therefore, one has to cross-examine it with the productivity value for a comprehensive scalability analysis. In our future work, we want to rectify this shortcoming. Our idea is that the scalability metric should not be a simple ratio of the two productivity values but it should be a function of their magnitude as well. We think that this can be achieved by defining a productivity metric for each topology as suggested in [165] and then modeling the scalability metric of the algorithm based on all these factors. Moreover, we would like to modify our simulation model in such a manner that we are able to run simulations under congested network traffic loads in large topologies.
We believe that the scalability framework presented in this chapter will help the designers of routing protocols to consider scalability as an important factor in their design and development, adding empirical validation as a corrective measure at an early stage.

# 5

# BeeHive in real networks of Linux routers

*The major contribution of the work presented in this chapter is a* Natural Engineering *approach that we developed to design and implement Nature inspired routing protocols in Linux routers. The approach helped in developing a* natural routing framework *inside the network stack of the Linux kernel. The* natural routing framework *was instrumental in realizing* BeeHive *inside the Linux kernel. We developed a protocol verification system to compare the performance of the algorithms from a simulated network with that of a real network. We performed extensive experiments in both environments to show that the performance of* BeeHive *is significantly better than* OSPF. *The work is a quantum leap because we have laid the ground for empirically refuting the deeply routed notion held by the engineers in the telecommunication industry that Nature inspired routing protocols are not economically viable because they cannot be implemented with the existing resources. The work, we believe, will also be instrumental in bridging the gap between different communities through the process of cross-fertilization of their design doctrines.*

## 5.1 Introduction

The work presented in this chapter was undertaken in order to respond to attitudes and prejudices from the classical Networking community. There, the belief is strongly held that Nature inspired routing protocols, though their intriguing character is recognized, can not provide benefits in real networks as they are only evaluated in simulations. The criticism is that these algorithms do not pay attention to engineering principles and constraints. Consequently, their installation on real world routers will require additional hardware and software resources. As a result, the benefit-to-cost ratio of the algorithms will render them economically nonviable to the telecommunication market where a cut-throat competition exists between different router vendors. Up to date, according to our knowledge, the lack of any empirical work showing the benefits of such algorithms in real networks has certainly helped to maintain and strength this attitude.

We agree with the engineers working in the telecommunication industry about the *current lack of an engineering vision during the design and development of Nature inspired routing protocols. AntNet* had been proposed eight years ago and *DGA* was proposed four years ago, and as of today, these significant and novel algorithms have not been implemented into real networks. So, their is not yet any commitment to accept Nature inspired routing algorithms in real networks. Given this, a particular effort is needed to overcome above-mentioned attitude that is widely spread in industry.

With this in mind, we put a strong emphasis on designing and developing an engineering approach during the design and development of our Nature inspired protocol. As a particular imperative we decided that *the algorithm must be realizable in existing real world routers, without the need of additional hardware or software resources to make it economically viable. The algorithm should not require any software components in the simulation that are not available in the real world networks.* We hold this as a basic principle for *Natural Engineering* (see Chapter 1). As a result,

this will lead to:

- developing economically viable Nature inspired networking systems, an aspect so far neglected by the Nature inspired routing community.
- ensuring that the algorithm developers avoid making unrealistic assumptions about the environment of real networks. In particular, algorithms must not utilize any features that are not available in real world routers.
- taking into account resource constraints in the real world, a fact that has so far received very little attention in Nature inspired routing community.
- demonstrating that Nature inspired routing protocols can truly deliver performance results in real networks similar to those obtained in a simulation environment. This will be instrumental for refuting the suspicion held in the commercial world. At any rate, *it will still contribute to a radical directional shift of design work in the Nature inspired routing community.*
- providing for cross-fertilization of ideas and paradigms from fundamentally different design approaches. This opens a good perspective in developing state-of-the-art networking systems for complex networks of the new millennium.

The major contribution of the work is the engineering approach mentioned above that we realized while developing our Nature inspired routing algorithm, *BeeHive*, to operate in packet switched telecommunication networks. We have implemented *BeeHive* inside the network stack of Linux operating system in a real world router. Using our comprehensive performance evaluation framework, we compared its performance with *OSPF*, in a real network of 8 Linux routers, part of our network systems lab of LS III at the University of Dortmund. The results obtained from the extensive experiments reveal that our engineering approach has paid its dividend because *BeeHive* outperforms *OSPF* in a real network of Linux routers even in those aspects that had been considered critical by the Network engineering community.

### 5.1.1 Organization of the chapter

Section 5.2 will provide an overview of different design approaches available for implementing a routing algorithm in a real Linux router. The section will also introduce the important engineering issues that the designer of a Nature inspired routing protocol should consider during the life cycle of a protocol. We will introduce our *natural routing framework* in Section 5.3 by highlighting its algorithmic-independent and the algorithmic-dependent components. The reader will appreciate that the framework is general enough to realize the relevant features of a Nature inspired routing protocol. Section 5.4 introduces our protocol verification framework that we developed to comprehensively compare and evaluate an algorithm both in a simulation environment and in a real network of Linux machines. We will introduce the motivation behind our experiment design in Section 5.5 and then discuss the experimental results. Finally, we provide a summary of the chapter.

## 5.2 Engineering of Nature inspired routing protocols

The engineering approach is of paramount importance for the realization of a routing protocol in a real network router [113]. Our extensive research into this intriguing domain has unfolded the true challenges that one faces in undertaking this strenuous task. We followed the basic software engineering design principle: *a good software organization emphasizes structures and components that are easily reusable. The reusability is achieved by decoupling their design from the implementation level details* [21, 33]. Such an approach enables a designer to concentrate on top-level design issues and pay little attention to the implementation details at an initial stage of the protocol development. Combined with our *Natural Engineering* approach, this helped in accomplishing the

challenging task of creating a *natural routing framework* inside the network stack of the Linux kernel. We addressed the following issues in the chronological order:

1. Structural design of a routing framework
2. Structural semantics of the network stack
3. System design issues

### 5.2.1 Structural design of a routing framework

The performance of a routing framework depends on its *structure* and cost of certain operating system (OS) functions, such as context switching and interrupt handling [39]. The *structure* of a framework defines the partitioning of its important components between the user space and the kernel space [110]. Consequently, one should carefully make this crucial decision because a thoughtless or imperfect partitioning of the components between the user space and the kernel space leads to frequent context-switchings resulting in a poor performance [110, 41]. A *structure* should be designed in such a manner that it carefully optimizes the access to the OS functions like process scheduling, memory allocation and management, and interrupt handling. The extensive studies done by Kay and Pasquale [108] indicate that the cost of processing an interrupt, copying a packet from the kernel space to the user space and context-switch overheads are at least as expensive as the protocol processing. A good routing framework should take all of these factors into account.
Thekkath et al. [197] have provided a good survey of the choices for partitioning the components of a protocol stack between the user space and the kernel space. However, their major emphasis is on structuring a *protocol stack* and not a *routing framework*. Nevertheless, the design principles and choices are valid for a *routing framework* as well. According to Keshave [110], a designer has to make a multi-way trade-off among the following factors in order to reach at a suitable decision on the partitioning problem:

- *Software engineering considerations* are influenced by the difficulty level of writing, testing and maintaining the code for the routing framework either in the user space or in the kernel space.
- *customization* is the ability of a routing framework to satisfy the requirements of the applications.
- *Security* concerns stem from the fact that a user process can harm other user processes. One has to be careful in placing a service into an untrusted user space as opposed to a trusted kernel space.
- *Performance* is the ability of a routing framework to accomplish its task as quickly as possible. The interested reader will find details about performance enhancement techniques/systems in [133, 69, 214, 108, 97].

In the following we introduce three basic choices: *a monolithic routing framework in the kernel space, a monolithic routing framework in user space, and a hybrid implementation in the kernel space and user space*. We considered the first one in our routing framework.

#### Monolithic routing framework in the kernel space

The most important components of a routing framework are: routing tables, route discovery and maintenance, and packet switching. In this approach, all of these components are encapsulated inside a module within the Linux kernel. Figure 5.1 illustrates this structure. This routing framework has the highest security and the best performance, however, it is an ostentatious task to implement a module inside the kernel and then debug the kernel code. The reasons are: the kernel

space has only a basic set of data structures that are significantly limited in their functionality. The important libraries like math, Standard Template Library (STL) etc. available in the user space are non-existent in the kernel space prolonging the implementation and debug phase. Last but not least, the routing framework is strongly coupled with the kernel code of a particular kernel version, and porting it to new enhanced kernel versions may not be straight forward, at times, which makes its source code difficult to maintain. The interested reader will find further details in [197, 214].

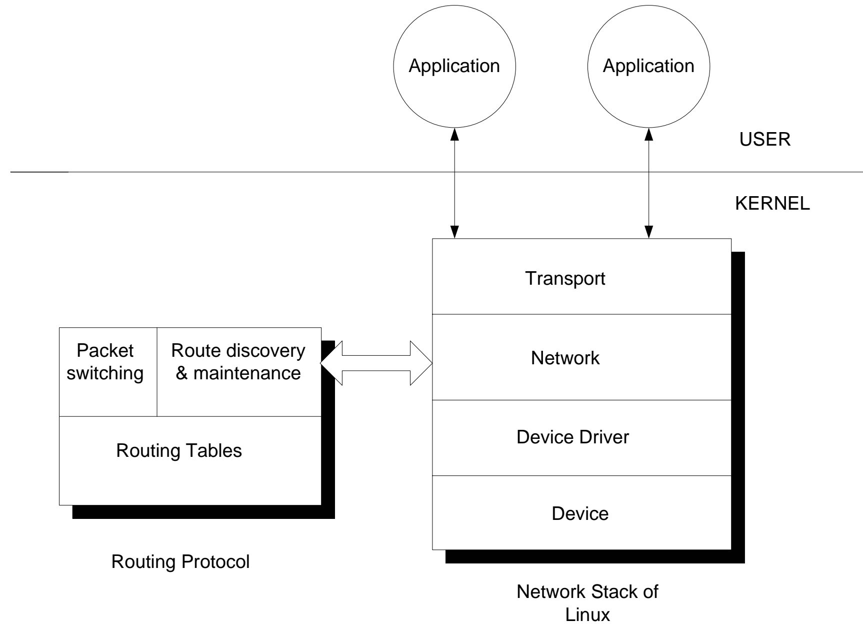



Figure 5.1: Monolithic implementation in kernel space

**Monolithic routing framework in user space**

In this approach, all components of a routing framework are implemented as a single process in the user space [197, 205, 71]. Figure 5.2 illustrates this structure. A user space module provides the flexibility of using user space data structures and libraries that significantly reduces the time of realizing a routing framework in a router. The debugging in user space is significantly easier and straightforward as compared to the kernel space. The routing framework is completely decoupled from the services/features of the kernel which significantly simplifies its code maintenance. However, the performance of the routing framework is poor because for each arrival of a packet, two context-switches are made: one from the kernel space to the user space to find the next hop leading toward the destination; another context-switch from the user space to the kernel space to enqueue the data packet at the network interface of the next hop. In case of agents, an additional copy of the agent is to be made from the kernel space to the user space in order to carry out its detailed processing in the user space. Finally, the agent is copied back from the user space to the kernel space. If the CPU scheduler does not give enough preference to the routing framework process then this scheduling latency degrades the over-all performance of the algorithm. Consequently, all of these factors significantly degrade the packet-switching rate of a network router. Last but not least, moving a routing framework into the user space has serious security concerns as well. However, the solutions to the security concerns are discussed in [69, 72, 197].

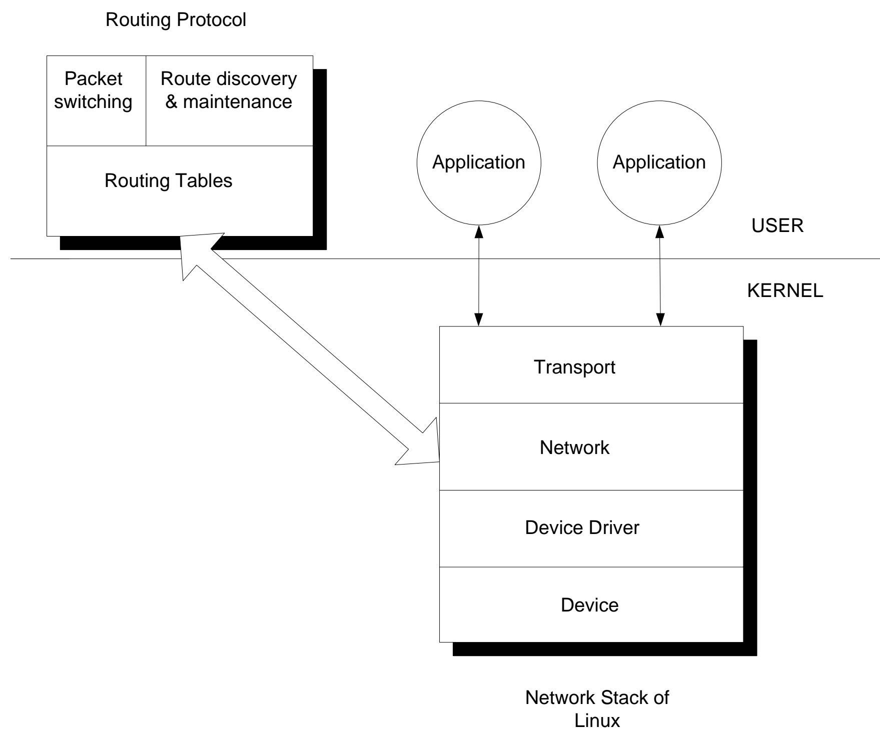



Figure 5.2: Monolithic implementation in user space

**Hybrid routing framework**

In this approach, the control path is separated from the data path. Consequently, the route discovery and maintenance component is moved into the user space while the routing tables and the packet-switching components remain in the kernel space. The two modules communicate with one another via the /proc file system [121]. This approach combines the benefits of easy implementation in the user space, of handling the complex behavior of agents (which requires special libraries and tools), with the efficient packet-switching of data packets in the kernel space, without the need of two context-switches for each routing decision. Figure 5.3 illustrates the approach. The frequency of agent-processing is significantly smaller as compared to packet-switching under high loads, therefore, the performance of the algorithm is not significantly degraded. The agent processing is the most important component of a Nature inspired routing algorithm and this approach completely decouples it from the corresponding kernel version, therefore, it becomes easier to maintain such a routing framework.

We decided to opt for the monolithic implementation in the kernel space after thoroughly review-

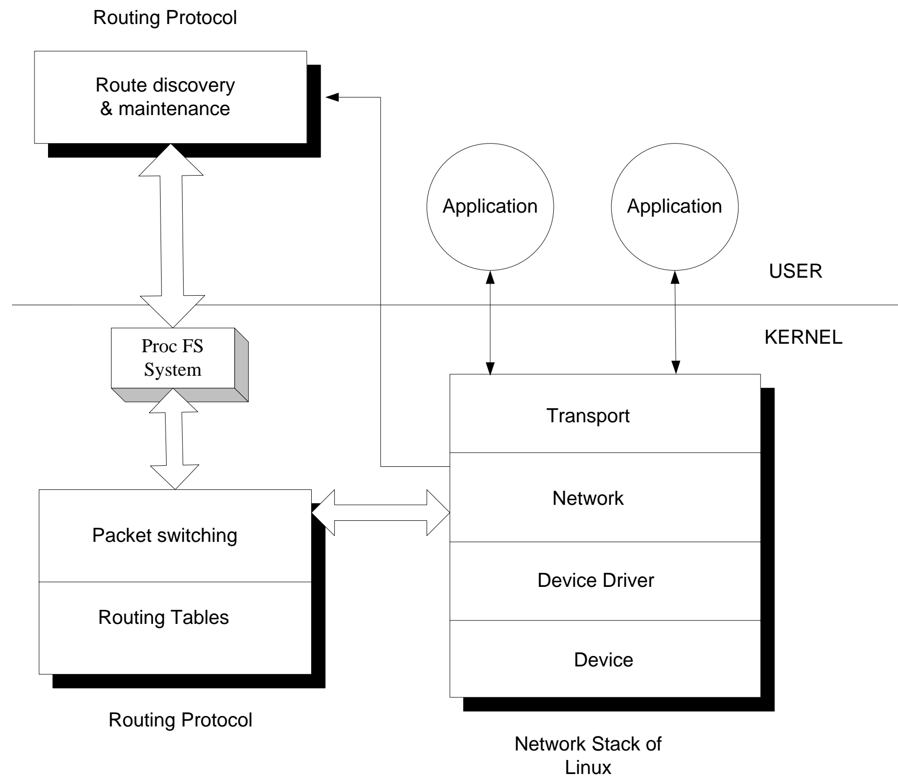



Figure 5.3: Hybrid implementation

ing the merits and demerits of the three approaches. The security and performance issues played a vital role in our decision making process.

## 5.2.2 Structural semantics of the network stack

The next issue that influences the design of a routing framework is the interface among different layers of a network protocol stack. Generally, three strategies are employed [110]:

1. The *Single context* approach calls for a single non-preemptible thread of execution in which only one packet is processed at a time, by locking the access to the stack. For such a system the only option available is monolithic implementation in the kernel space.

2. The *Tasks-oriented model* implements each layer as a task which is scheduled by a central task scheduler with a pointer to the buffer containing a packet. In this approach a packet is shepherded through the stack by a sequence of tasks that mutually schedule each other. Such an approach allows for Quality of Service (QoS) guarantees because a scheduler has the freedom to schedule high priority tasks over low priority tasks [5]. However, this approach has a higher latency because of frequent intervention of the scheduler during the journey of a packet through the stack. For such a system, any of the above-mentioned three strategies will work.

3. The *Upcall* architecture associates a thread with each data packet, and it is responsible for all layers of the protocol processing. High priority threads handle high priority packets. Each layer registers send and receive entry points with its lower layer. As a result, it can make a procedure call on the preregistered entry point [40, 97] during the reception or transmission of the packets. In such a model, no data is copied between the layers that enhances its performance. Linux has got an upcall architecture, therefore, it is important to identify the entry points to/from the data link layer and to/from the transport layer. For such a system, any of the above-mentioned three implementation strategies can be utilized.

The upcall system architecture facilitates to conveniently structure actions of a routing protocol in an asynchronous manner, as responses to the five basic events shown in Figure 5.4 through entry/exit points. The NetFilter architecture in Linux [156, 1], based on the concept of Packet

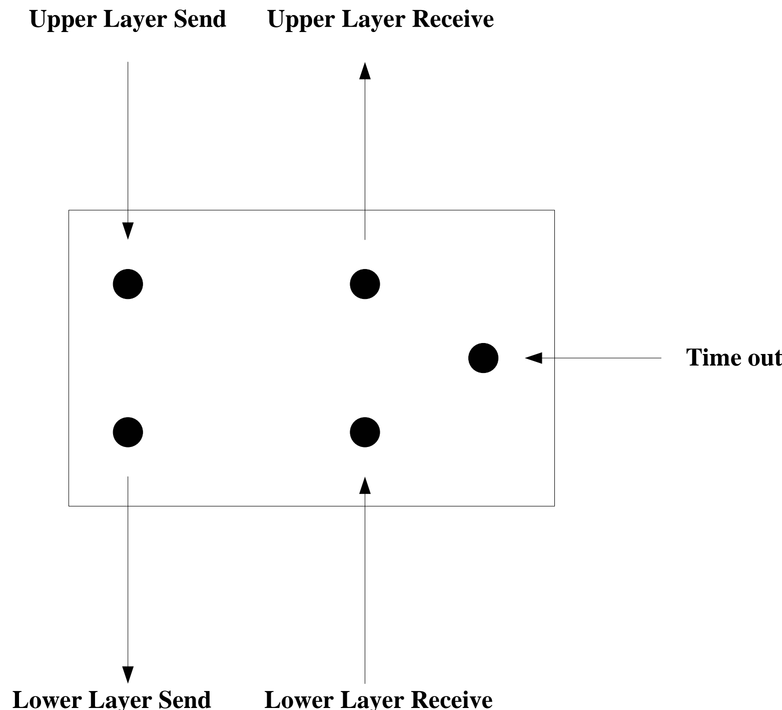



Figure 5.4: Protocol block from [110]

Filter [134], provides a simple mechanism for receiving and processing packets from the protocol stack in the network layer. This helps in realizing a routing framework as a protocol block as illustrated in Figure 5.4.

### 5.2.3 System design issues

A routing framework is part of a distributed system which is built from hardware and software resources [110]. The development of a routing framework, therefore, requires a thorough understanding of system design techniques with a high emphasis on computer networks. An interested reader will find a detailed treatment on operating systems in [173, 195], on computer architecture in [92], and on network systems/protocols in Unix in [187, 186, 184, 240, 185]. We, as mentioned before, followed an engineering approach, which tries to optimally balance the requirements for *constrained* and *unconstrained* resources, during the design and development of a routing framework. We tried to account for the following resources: time, space, computation and labor.

1. *Time:* Our routing framework must be simple enough so that it requires as little time of the processor as possible. The framework must not put any additional hardware constraints on the existing computers for its functionality. This will also decrease its time to market as well.

2. *Space:* Like time, space is also a limited resource. Space constraints are expressed by limits on available memory, data packets in routers, routing tables and other routing information obtained from the agents. A routing algorithm should try to utilize this resource as efficiently as possible. A carefully designed and efficiently implemented routing framework is a prerequisite for achieving this goal.

3. *Computation:* Our framework must be able to work with the low-end Pentium III series of processors that we have at our disposal in our LS III network systems lab. This will refute the popular notion held by the networking system community that Nature inspired approaches are too complex to be realized with the existing hardware and software resources.

4. *Labor:* Our framework should be designed in such a fashion that we can efficiently implement it with the help of two full-time software developers working for 8 months. The workload initially estimated was about 2800 man/machine hours. Luckily, we were able to accomplish the task in approximately 3000 man/machine hours. A well planned design strategy, as discussed in the previous section, helped in meeting important milestones and deadlines. In the first phase, we implemented and tested the *BeeHive* algorithm in a virtual network of virtual Linux machines [244], and in the second phase we made the final transition of realizing the algorithm in a real network of real Linux machines [88].

We agreed to work on a design model that is scalable: it can be easily adapted to new architectures, systems, and increasing network size. We expect that this feature will significantly reduce the cost of porting the framework to large heterogeneous systems.

## 5.3 Natural routing framework: design and implementation

We designed, developed and implemented our *natural routing framework* with two layers of abstraction: algorithmic-independent and algorithmic-dependent. The algorithmic-independent framework consists of structures and services that facilitate the realization of a Nature inspired routing protocol inside the network stack of Linux kernel, by utilizing its interfaces and the services it offers. As mentioned in the previous section, typically a Nature inspired routing protocol consists of three components:

1. *Agents* which collect the information about the state of the network and then store it at the nodes they visit.

2. *Routing tables* that act as the repository for storing the routing information collected by the agents. The tables also cater for information exchange, either directly or indirectly, through the environment. The Nature inspired routing protocols require specialized tables for their correct functioning.

3. *Packet switching* task is achieved by distributing data packets in a stochastic manner as per quality, which is calculated based on the information in the routing tables along the paths.

An algorithmic-independent framework consists of structures and features that contribute to implementing the above-mentioned components of a routing algorithm.

### 5.3.1 Algorithm-independent framework

The algorithm independent framework consists of an algorithmic module inside the Linux kernel. The motivation for a monolithic implementation in the kernel space has been comprehensively substantiated in the previous section. The framework consists of three components: Agent manager, Agent processor and Packet processor (both for incoming and outgoing data packets). For these components, we now discuss in detail the implementation of our module.

#### Agent manager

The task of the agent manager is to periodically or aperiodically launch the agents which collect the routing information from the network. The agents can be launched either in a broadcast mode, or in a point-to-point mode, depending on the design of the algorithm. The manager sleeps in an *interruptible* fashion after launching the agents. The *interruptibility* enables the manager to asynchronously/aperiodically launch the agents if an event of interest occurs.

#### Agent processor

The agent processor runs in the background as a *daemon* and is responsible for receiving and processing the agents. Once it receives an agent it executes the behavior component of the agent. Subsequently, if the agent needs to update/modify the routing information in the routing tables then it can request the agent processor to do it, which is the only component authorized to alter/update the routing table. Once the agent has executed its actions, the agent processor forwards it, if needed, to all neighbor nodes of the current node (if broadcast mode is selected) except from which it arrived. Otherwise, in the point-to-point mode, it will forward the agent to the next hop that is determined by the routing algorithm.

#### Packet processor

The packet processor is allowed to access the routing information in the routing tables but cannot modify it. A packet processor can receive packets either in its *pre_input_hook* that in turn is connected to the IP_PRE_ROUTING of the NetFilter, represented as "PRE Mangle" in Figure 5.5, or in its *output_hook* that in turn is connected to IP_LOCAL_OUTPUT, represented as "OUT Mangle" in Figure 5.5. The packets passing through the current node are processed in the *pre_input_hook*. If the current node is the destination of the packet then it is delivered to the transport layer otherwise it will be forwarded to the next hop interface, after consulting the routing table. The packets that are generated at the current node are processed in the *output_hook*, and the routing decision is made in this hook.

### 5.3.2 Algorithmic-dependent BeeHive module

The *BeeHive* module utilizes the services and interfaces of the above-mentioned algorithmic-independent routing framework. The tasks of processing and managing the information of the *bee agents* is accomplished by utilizing the agent manager and agent processor components of the framework. The packet switching functionality is achieved through utilizing the services of the packet processor, and the routing information is maintained in the routing tables that are especially designed and developed for multi-path stochastic routing algorithms. Figure 5.6 shows in detail the block diagram of the *BeeHive* module.

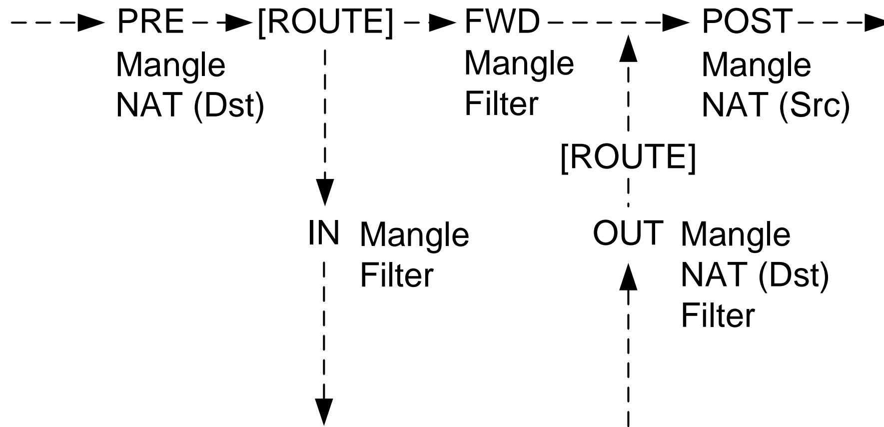



Figure 5.5: NetFilter hooks

**Bee processing and management**

The important task of the *agent manager* is to periodically or aperiodically launch a *bee agent* as depicted in the procedure *launchBeeAgents(s,*$n_i$*,n)* in Algorithm 1 (please refer to Chapter 3). The *bee agents* can also be asynchronously launched once a node receives m (currently 240) packets; the paths leading to the destination node of frequent network traffic should be sampled and explored at a higher frequency than the others. The agent manager, once awake, does the following three tasks:

- It creates a *bee agent* and writes its own IP address into the source field. It assigns it a bee agent id and a time to live value (TTL) in term of the number of hops permitted.
- It retrieves the IP address of each valid network interface except of the loopback adapter. It creates a replica of the *bee agent* for each valid IP interface and assigns it a distinct replica id.
- It increments the bee agent id.

The *bee agents* launched by the *agent managers* are received at other nodes through the *agent processors*. The *agent processor*, as discussed earlier, is a daemon process that listens on a predefined UDP port, which is reserved for the transmission and reception of *bee agents*. The tasks of an *agent processor* are outlined in the *processBeeAgents()* method of Algorithm 1 (please refer to Chapter 3). The concepts of *foraging regions* and *foraging zones* are not implemented because their benefits on a small topology are not obvious. The *agent processor* does the following tasks:

- If time to live timer (TTL) has reached zero or if the replica already visited the same node then it is killed (to avoid loops).
- At start up time, the *agent processor* estimates the propagation delay of the transmission links connected to its neighbors. The received replica updates its estimate of the queuing delay by calculating the queue length of the interface of the neighbor, from where it arrived, at the current node. Similarly, it updates its estimate of the propagation delay.
- The received replica then requests the *agent processor* to update, in the routing tables, the estimates of the queuing and propagation delays of its source node for the neighbor from where it arrived. Please remember that only an *agent processor* has the permission to modify the routing tables.
- If available, the replica reads the estimates of the queuing and propagation delays of other replicas that are also launched from its source node. The replica updates its view of the network state based on the information communicated by other replicas (see Figure 3.3).

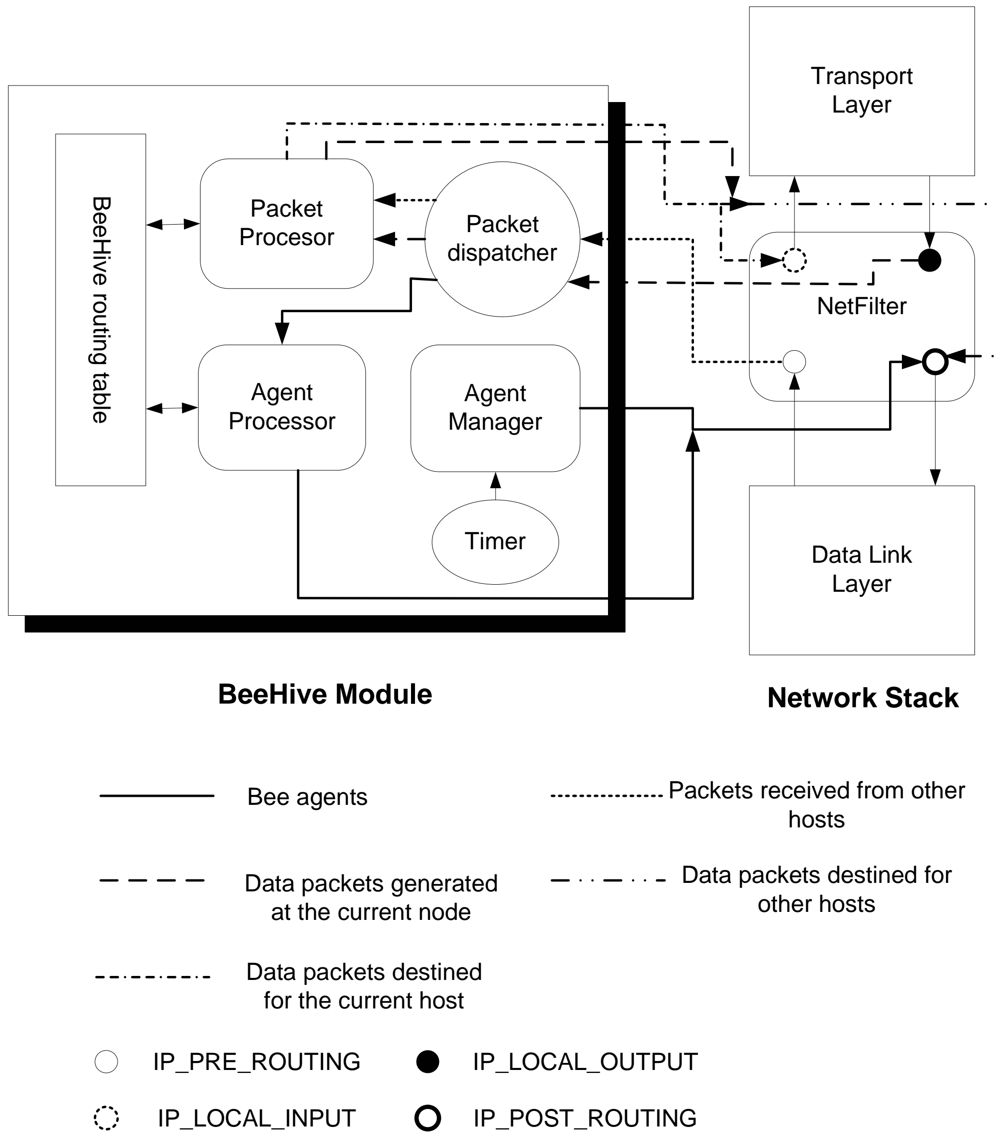



Figure 5.6: Block diagram of BeeHive module

- If a replica of the same generation (bee agent id) had already been received at the node then the *agent processor* kills the current replica.
- The *agent processor* retrieves the IP address of each valid network interface except the loopback adapter and the network interface from where the replica is received. The replica is broadcast to the retrieved IP addresses without modifying its replica or agent id.

**Packet processor**

The data packets are sent to the *packet processor* by the *packet dispatcher*, which is connected to NetFilter at two interfaces: *pre_input_hook* and *out_hook*. The packets generated by the applications running at the current host are received in the *out_hook*. The packets sent from the network interface card are received in the *pre_input_hook*. The general task of a packet processor is to access the information in the routing tables, calculate the quality of each neighbor that can be the next hop towards the destination, and then stochastically choose the next hop toward the destination.
The floating point operations are not supported in the Linux kernel, therefore, one has to adapt the packet switching algorithm such that it works with integer numbers only. The simple solution is to rescale the probabilities in the range from 0 to 100 instead of 0 to 1. We simply rescale $g = \frac{K}{p+q}$ where K is a large integer value, p is the propagation delay and q is the queuing delay. Finally the integer probability $\theta_{jd}$ is $\theta_{jd} = \frac{g_{jd} \times 100}{\sum_{k=1}^{N} g_{kd}}$, where N is the number of the neighbors of the current node.
The *packet processor* can receive a data packet from the network interface card through *pre_input_hook* of *packet dispatcher*. If the destination of the packet is the current node then it is given to the upper layers via *IP_LOCAL_INPUT* hook of NetFilter, represented by "IN Mangle" in Figure 5.5, otherwise the function *b_route_input()* is called. This function accesses the bee routing table and applies the above-mentioned algorithm to find a next hop. The Linux kernel caches the routing tables for efficient routing of the data packets, under the assumption that the quality of the routes will not drastically change in small intervals of time. In the *b_route_input()* function, the complete processing relating to routing is done, and finally *ip_route_input()* of the network stack is called by giving it the next hop as the final destination. This simple circumvention makes Linux access its cached routing tables with the next hop as the final destination. As a result, the packet will be queued in the network interface of the next hop host that is determined by the *BeeHive* algorithm. The *packet processor* calls the *ip_route_output_flow_output()* function for processing the data packets that have been received through *out_hook* of *packet dispatcher*. These packets are generated at the current node. The processing related to routing is quite similar as discussed in the *pre_input_hook* case.

**Bee routing tables**

The routing tables of the Linux kernel render themselves useless once it comes to supporting Nature inspired routing protocols because of the following reasons:

- Nature inspired routing algorithms take a routing decision on a packet basis. Therefore the idea of caching based on session, destination IP, source IP, port and type of service (TOS) values does not significantly help.
- Nature inspired routing algorithms are multi-path routing algorithms. Consequently, it is possible to reach each destination host through multiple neighbors and this information has to be incorporated in the routing tables.
- Nature inspired routing protocols implicitly assume that the hosts are *polymorphic*. A polymorphic host has different IP addresses for each of its network cards but all of these IP addresses represent the same host. *OSPF* does not care about this because it finds a single path to each IP address.

These shortcomings motivated us to design and develop a *BeeHive* specific routing table that would not suffer from the above-mentioned limitations. The bee routing table consists of three further tables as shown in Figure 5.7: IP table, host table and neighbor table. The IP table contains all the possible destination IP addresses. Each IP address is mapped to a host in host table. If a host has multiple IP addresses then the corresponding IP entries point to the same host. Each host points to a neighbor table that contains all neighbors that could lead to the same destination. The tables are currently implemented with the help of linked list structures. The IP table consists of

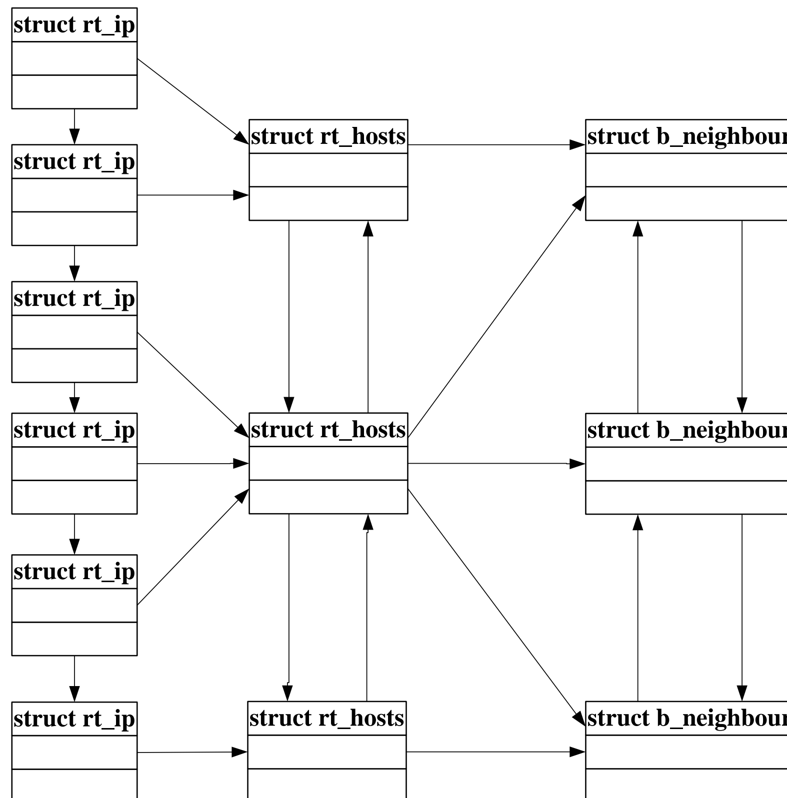



Figure 5.7: Top level routing table

a linked list of rt_ip structures as shown in Figure 5.8. Each entry in the IP table is linked to an entry of hosts table, which is a double linked list of rt_hosts structures (see Figure 5.8). The basic idea is that different rt_ip structures (IP addresses) of the same host point to the same rt_hosts structure, and a rt_hosts structure points to all of its rt_ip structures. We maintain a double linked list of b_neighbour structures (as shown in Figure 5.9) that is only accessible through an entry of a linked list of rt_neighbour structures (as shown in Figure 5.10). Each entry in the double linked list of rt_hosts (Host table) points to its own linked list of the rt_neighbour structures through which a packet can reach its destination node. In order to efficiently implement the routing table, the linked lists of rt_neighbour structures of different hosts point to their corresponding entries in the same double linked list of b_neigbhour structures. This significantly improves the fault management; if a neighbor node has crashed then its corresponding b_neigbour entry is made invalid and this virtually removes it from the linked list of all rt_neighbour structures because an invalid entry is not considered in the routing process. Please refer to Table 5.1 for the description

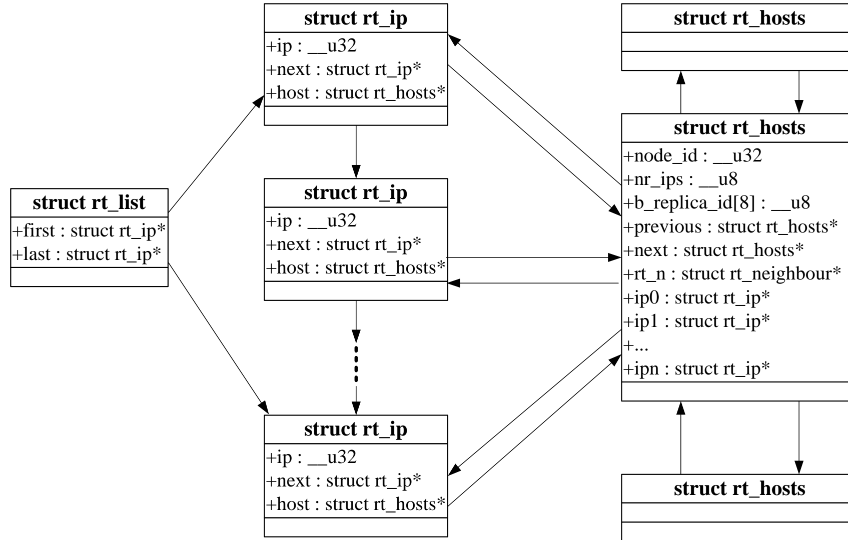



Figure 5.8: IPs table

of important symbols used in rest of the chapter.

## 5.4 Protocol verification framework

We have designed and developed a comprehensive and sophisticated protocol verification framework for extensive performance evaluation, comparison and verification of a routing protocol, both in a simulation environment and in real networks. The conceptual block diagram of the framework is illustrated in Figure 5.11.

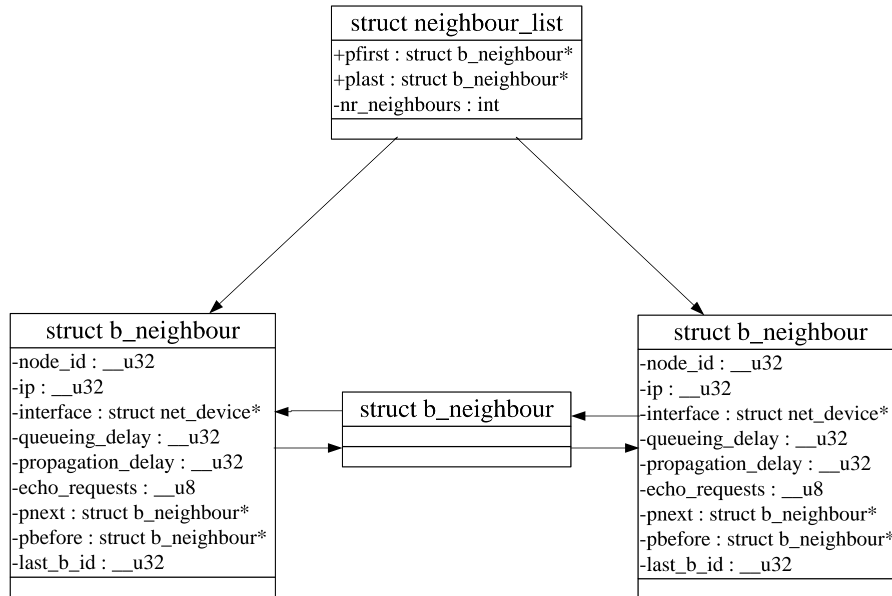



Figure 5.9: Neighbors table

The framework consists of the following important building blocks:

1. The *traffic generator* is a tool that generates arbitrary but realistic traffic patterns which represent different classes and types of the real network traffic. The traffic generator has been introduced in Chapter 3.

2. The *performance evaluation framework* is a tool that calculates the relevant performance values from the experiments. The framework is comprehensively introduced in [216]. The framework is also described in Chapter 3

3. The *results database and plotter utility* is a tool that stores the most important parameters from the experiments in a database. The plotter utility consists of a set of scripts written in Perl that reads the results from the database and then presents them, as per choice of the user, either graphically or in a table.

We have implemented the traffic generator and performance evaluation framework both in the network simulator OMNeT++ [203] and in an application layer of the Linux operating system network stack as illustrated in Figure 5.11. By assigning the same values to the above-mentioned parameters of the traffic generator, we expect to generate nearly similar traffic patterns in a simulation and real network environments. In the next step of our protocol verification process, we utilize a topology generator that generates a topology of a given amount of nodes and links. The links are fully characterized by their bandwidth, propagation delay and bit error rate. During this process, we select a topology that we will use in our simulation environment and the real network of Linux routers. The topology is known as simpleNet and it is shown in Figure 5.12. The detailed description of each Linux router machine that includes its processor, RAM, cache, hard disk, and the network interface cards is cataloged in Table 5.2.

The protocol verification principle is: *if we generate same traffic patterns through same traffic generators both in OMNeT++ and the Linux network and utilize the same performance evaluation framework again both in OMNeT++ and the Linux network then the performance values obtained from the simulation environment should be consistent to the ones obtained from the real Linux network with minor deviations, provided our simulation environment depicts a somewhat realistic*

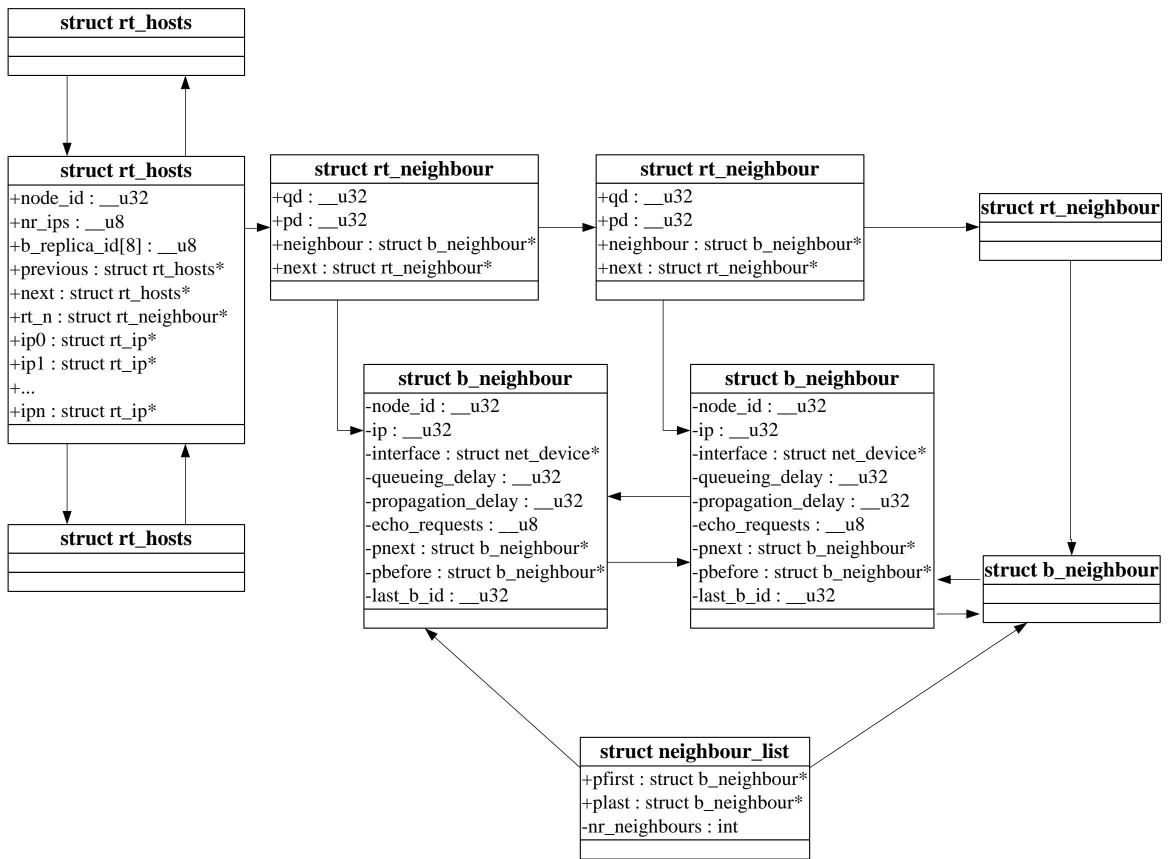



Figure 5.10: Detailed bee routing table

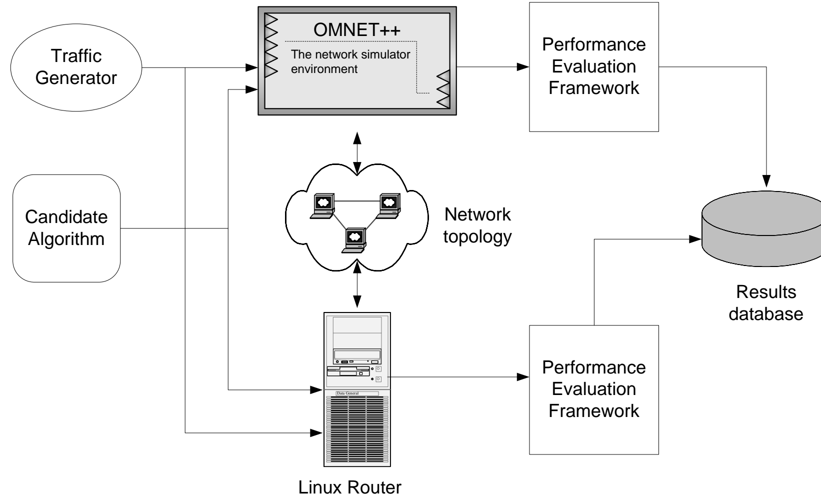
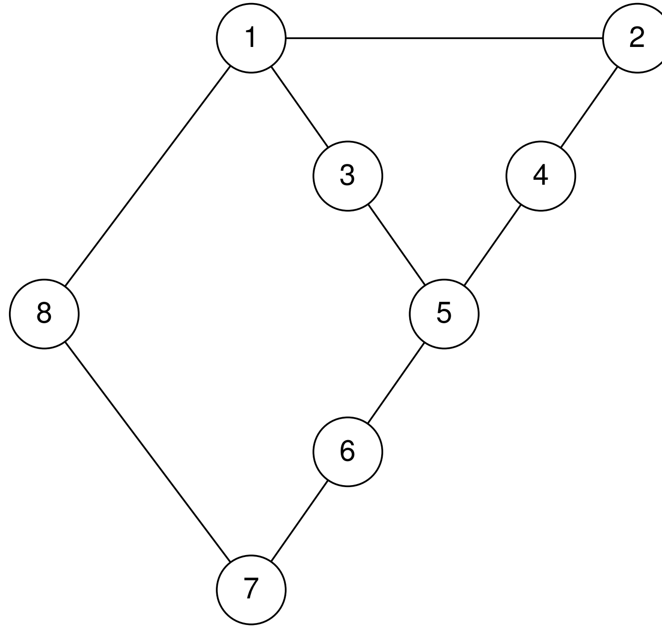


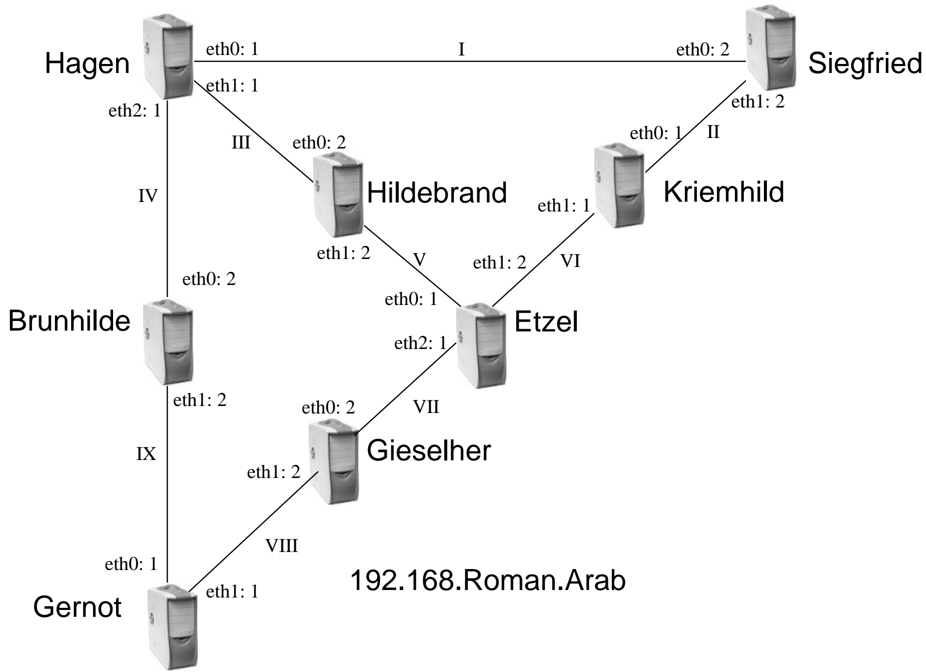

Figure 5.11: Protocol verification framework

| | |
|---|---|
| $P_{gen}$ | Total number of packets generated |
| $P_{rec}$ | Total number of packets received |
| $S_{to}$ | Total number of sessions started |
| $S_{co}$ | Total number of sessions completed |
| $T_{av}$ | Average throughput (M/sec) |
| $P_d$ | Packet delivery ratio (%) |
| sSize | size of a session (in bytes) |
| pSize | size of a packet (in bytes) |
| MSIA | Mean of session inter-arrival times (sec) |
| $R_o$ | Control overhead (%) |
| MPIA | Means of packet inter-arrival times (sec) |
| $S_o$ | Suboptimal overhead (%) |
| $P_{loop}$ | Percentage of packets that followed a cyclic path |
| $S_c$ | Session completion ratio (%) |
| $t_d$ | Average packet delay (msec) |
| $t_{90d}$ | 90th percentile of packet delays (msec) |
| $S_d$ | Average session delay (msec) |
| $h_o^{sd}$ | minimum hops needed to reach from node s to node d |
| $S_{90d}$ | 90th percentile of session delays (msec) |
| $h_i^{sd}$ | hops packet i took to reach from node s to node d |
| $J_d$ | Average jitter value (msec) |
| $h_{av}$ | Average hops count the data packets |
| $J_{90d}$ | 90th percentile of jitter times (msec) |
| $\beta_c$ | Queue Length (in packets) |

Table 5.1: Symbols used in the chapter




(a) simpleNet in OMNeT++




(b) simpleNet of Linux machines in LS III lab

Figure 5.12: simpleNet topology: 8 routers, 9 bidirectional links each of 10 Mbits/sec

| Node in OMNet++ | Node in Linux | Machine specification | Address in OMNeT++ | Address in Linux |
|---|---|---|---|---|
| 1 | Hagen | Intel Pentium III, 500 MHz, 512KB L2 Cache, 128MB RAM, 3 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 1 | 192.168.1.1<br>192.168.3.1<br>192.168.4.1 |
| 2 | Siegfried | Intel Pentium III, 500 MHz, 512KB L2 Cache, 128MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 2 | 192.168.1.2<br>192.168.2.2 |
| 3 | Hildebrand | Intel Pentium III, 500 MHz, 512KB L2 Cache, 384MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 3 | 192.168.3.2<br>192.168.5.2 |
| 4 | Kriemhild | Intel Pentium III, 500 MHz, 512KB L2 Cache, 384MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 4 | 192.168.2.1<br>192.168.6.1 |
| 5 | Etzel | Intel Pentium III, 500 MHz, 512KB L2 Cache, 128MB RAM, 3 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 5 | 192.168.5.1<br>192.168.6.2<br>192.168.7.1 |
| 6 | Giselher | Intel Pentium III, 500 MHz, 512KB L2 Cache, 384MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 6 | 192.168.7.2<br>192.168.8.2 |
| 7 | Gernot | Intel Pentium III, 500 MHz, 512KB L2 Cache, 384MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 7 | 192.168.8.1<br>192.168.9.1 |
| 8 | Brunhilde | Intel Pentium III, 500 MHz, 512KB L2 Cache, 384MB RAM, 2 Realtek Semiconductor Co., Ltd. RTL 8139/8139C+(rev10) network cards | 8 | 192.168.4.2<br>192.168.9.2 |

Table 5.2: The mapping of Hosts to IP Addresses in SimpleNet

*picture of a true network.* The deviation in performance values might stem from the differences of the simulation environment from a true Linux network topology such as:

- The clocks in OMNeT++ are perfectly synchronized. But in a LAN of Linux machines they are synchronized using Network Time Protocol (NTP) [131] that can provide an accuracy of approximately 10 milli-seconds.
- The network stack processing time during the processing of a packet, both control and data, is ignored in the simulation environment.
- The context-switching time from kernel space to user space and vice versa is ignored in a network simulator.
- The latency of scheduling a task, which is dependent on the scheduler of an operating system, is ignored in the simulation. This is of primary importance once multiple and concurrent sessions are active in a Linux router.
- The non-availability of high resolution timers in Linux [121] might result in MPIA values different from OMNeT++.
- The protocol-specific characteristics of Ethernet employed at the data link layer (layer 2), and its direct influence on the signals at physical layer, and also the transmission medium are ignored in the simulation for the sake of simplicity.

## 5.5 The motivation behind design and structure of experiments

The basic motivation behind designing the scenarios for the experiments was to cover a broad range of operational environments, and to quantify the performance values of different parameters in a systematic fashion. After extensive brainstorming, we agreed to divide our experiments into

three broad categories according to the network traffic: quantum traffic engineering, real world applications and hybrid traffic engineering. The idea of *quantum traffic engineering* is to provide to each tested algorithm, abstract but repeatable traffic patterns in order to obtain statistically significant performance values through multiple (in our case 10) independent attempts. We utilized our Scientific Quantum Traffic Generator (SQTG) introduced in Chapter 3 for this purpose. For real world applications traffic, we used the File Transfer Protocol (FTP) [155] to download large files. These experiments revealed quantifiable benefits, which are of pertinent interest, to the industry, in employing different routing protocols. Finally we utilized the Distributed Internet Traffic Generator (D-ITG) [22, 9, 10, 11, 8] developed at the *Universita' degli Studi di Napoli* in Italy to emulate Voice over IP (VoIP) traffic [28, 4] along with UDP traffic. Moreover, utilizing a traffic generator developed by a third party would eliminate any bias that might have been introduced to our *BeeHive* algorithm by our own traffic generator. We call these experiments hybrid ones because they simulate a real world application (VoIP) traffic but we are not using a true VoIP application . During the extensive performance evaluation and verification cycle, all performance values are an average of performance values obtained from ten independent experiments. This was done to factor out stochastic elements in the network environment or in the algorithms. The experiments were conducted for 1000 seconds, a good enough time for factoring out transients.

## 5.6 Discussion of the results from the experiments

In this section, we discuss in detail the results obtained from our extensive experiments, both in the simulation and in real networks. We discuss the results according to their category type as introduced in the previous section.

### 5.6.1 Quantum traffic engineering

We utilized our SQTG for this series of the experiments. We conducted 10 experiments by varying values of different parameters of the traffic generator. The parameter values are shown in Table 5.3. We generated UDP [153] traffic because it supports an unbiased evaluation of a routing protocol, in contrast to TCP [154], which has a complex mechanism to adaptively control the size of the congestion window if packets are lost.

| Exp | sSize | pSize | MSIA | MPIA | $\beta_c$ | Source | Destination | Hot Spot | Routers Down |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2130000 | 512 | 1.8 | 0.005 | 3000 | Hagen | Gernot | none | none |
| 2 | 2130000 | 512 | 1.2 | 0.005 | 3000 | Hagen | Gernot | none | none |
| 3 | 2130000 | 512 | 1.0 | 0.005 | 3000 | Hagen | Gernot | none | none |
| 4 | 2130000 | 512 | 0.9 | 0.005 | 3000 | Hagen | Gernot | none | none |
| 5 | 1065000 | 1024 | 1.8 | 0.005 | 1500 | Hagen | Gernot | none | none |
| 6 | 1065000 | 1024 | 1.2 | 0.005 | 1500 | Hagen | Gernot | none | none |
| 7 | 1065000 | 1024 | 1.0 | 0.005 | 1500 | Hagen | Gernot | none | none |
| 8 | 4055000 | 512 | 1.15 | 0.005 | 3000 | Etzel | Hagen | none | none |
| 9 | 1270000 | 512 | 1.15 | 0.005 | 3000 | Etzel | Hagen | Hagen(100,800)(0.001) | none |
| 10 | 2580000 | 512 | 1.15 | 0.005 | 3000 | Etzel | Hagen | none | Hilderbrand(300,600) |

Table 5.3: Parameters for traffic generator for Experiments 1 to 10

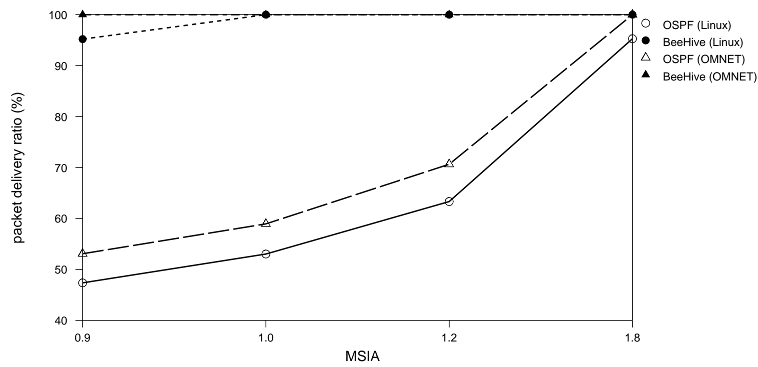
#### Experiments 1,2,3,4

The purpose of these set of experiments is to study the behavior of the algorithms under saturated network loads. The MSIA value was decreased from 2.0 sec to 0.9 sec in these experiments. The size of the session was kept constant at 2130000 bytes and the packets were sent from Hagen to Gernot (see Figure 5.12(b)). The experiments with the same parameters were conducted both in OMNeT++ and the real network topology. Figure 5.13 shows the packet delivery ratio and 90th percentile of packet delays distribution obtained both from OMNeT++ and Linux. As expected, the packet delivery ratio of *BeeHive* scales better than *OSPF* both in OMNeT++ and Linux. The

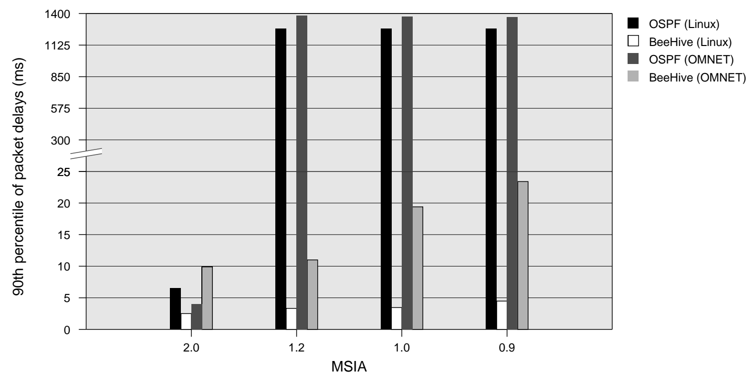
packet delivery ratio of *OSPF* drops significantly from 99.9% to about 50% as the MSIA value is decreased from 2.0 sec to 0.9 sec respectively. The reason is obvious: *OSPF* always utilizes the single path Hagen-Brunhilde-Gernot while *BeeHive* distributes packets over two paths Hagen-Brunhilde-Gernot and Hagen-Hildebrand-Etzel-Gieselher-Gernot. Consequently, *BeeHive* achieves 17 Mbits/sec throughput (see Table 5.4), which is approximately twice than the one achieved by *OSPF*. *OSPF* saturates the queues of path Hagen-Brunhilde-Gernot, as a result, a significant queue delay is experienced by data packets. Consequently, one can see from Figure 5.13(b) that the 90th percentile of packet delays of *OSPF* is significantly higher (1100 msec) than of *BeeHive* (20-25 msec). A reader might find an anomaly in Figure 5.13(b) because it suggests that the packet delays of the algorithms obtained from a real Linux topology are smaller (approximately 10 msec) as compared to the ones obtained from OMNeT++. Our investigation revealed that it is the result of a clock synchronization problem. Please remember that NTP, as discussed before, can provide a resolution of 10 msec only. Consequently, the packet delay in simulation and real networks might differ in the 10-15 msec range.
The effect of saturated network load is more prominent on the session completion ratio as shown in Figure 5.14(a). *OSPF* is hardly able to complete any session at a MSIA value of 1.2 sec or below. Please note in Figure 5.14(b) that session delay of *BeeHive* is significantly better than that of *OSPF*, both in OMNeT++ and simulation. However, the difference in session delay values obtained from the simulation and the real network is significant. We investigated the problem and found that non-availability of high resolution timers in Linux [121] is responsible for it. Our MPIA value is 5 msec while Linux has a scheduling uncertainty of 10 msec. As a result, a session takes longer to finish at the source, and this explains the larger session delay for the algorithms in Linux as compared with OMNeT++. With a MSIA value of 0.9 sec the resources of Linux machines were nearly consumed and that is why the performance values differ significantly from the simulation values, mainly because of a significant difference in the operational environment of simulation and reality. We have collected the important performance values form Linux and OMNeT++ environments in Table 5.4. Please note that most of the parameter values, with few exceptions, are easily traceable between simulation and reality. One can safely comment: *the relative tendency of the performance values of the two algorithms in both environments is approximately similar.*

**Experiments 5,6,7**

The tolerable error which stems from the performance values of the previous experiments led us to test the algorithms with less challenging traffic patterns. Therefore, we decreased the size of a session to 1065000 bytes and increased the size of the packet to 1024 byte. The larger packet size will ensure that a session finishes faster than the scenarios of the previous subsection. A smaller session delay will decrease the amount of the sessions that are concurrently active at the source node. Consequently, it will reduce the error stemming from the scheduling latency of the Linux scheduler. We decreased the MSIA value from 1.8 sec to 1.0 sec. We hoped to see a better relatedness of the performance values obtained from OMNeT++ to Linux topology. Figure 5.15(a) shows the packet delivery ratio of *OSPF* and *BeeHive*, respectively, obtained both from simulations and the real network topology. Both algorithms were able to deliver approximately all of packets at their destination under such a small network load. However, the 90th percentile of the packet delays of *BeeHive* is significantly better than *OSPF* because *BeeHive* distributes the traffic load on multiple paths and this results in significantly smaller queue lengths as compared with *OSPF*. Both algorithms are able to complete approximately all of sessions (see Figure 5.16(a)), and with approximately the same session delay (see Figure 5.16(b)). Please note that the difference in session delays obtained from the simulations and the real network topology is about 2 seconds which is reasonably acceptable and supports our thesis: *a concurrent number of sessions have to be scheduled by a Linux scheduler, and because of ambiguity in the scheduling latency the traffic patterns could differ from the ones in the simulation.* The other important performance values are collected in Table 5.5.

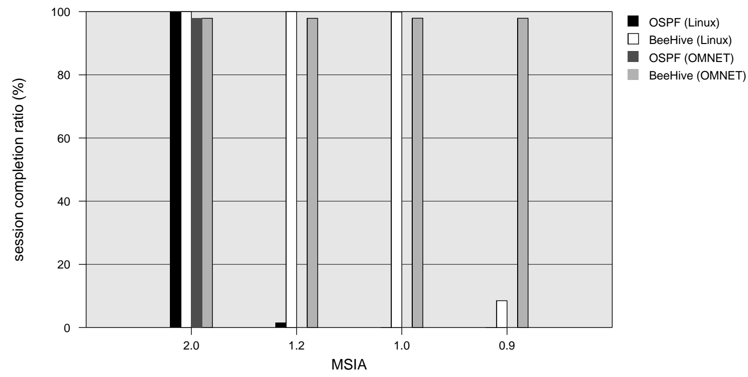



(a) Packet delivery ratio

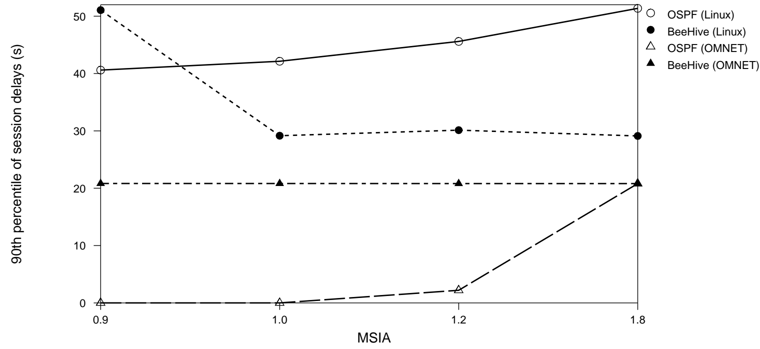



(b) 90th percentile of packet delays

Figure 5.13: Experiments 1-4 (packet delivery ratio and packet delay)




(a) Session completion ratio




(b) 90th percentile of session Delays

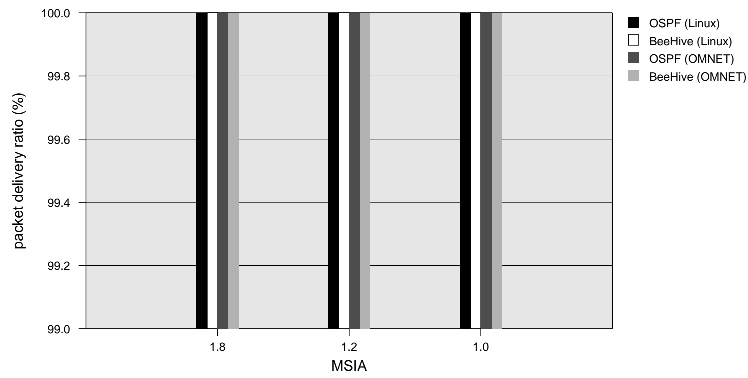
Figure 5.14: Experiments 1-4 (session completion ratio and session delay)

| Algorithm | | BeeHive | | | | OSPF | | | |
|---|---|---|---|---|---|---|---|---|---|
| Environment | MSIA | 2.0 | 1.2 | 1.0 | 0.9 | 2.0 | 1.2 | 1.0 | 0.9 |
| LINUX | $P_{gen}$ | 2040050 | 3402747 | 4082759 | 4497561 | 2056660 | 3382901 | 4059611 | 4511485 |
| | $P_{rec}$ | 2040050 | 3402747 | 4082667 | 4274467 | 2056660 | 2144274 | 2155929 | 2157964 |
| | $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $S_{to}$ | 500 | 830 | 995 | 1108 | 502 | 843 | 996 | 1107 |
| | $S_{co}$ | 500 | 830 | 994 | 94 | 502 | 13 | 0 | 0 |
| | $S_o$ | 2.785 | 7.961 | 9.261 | 9.706 | 0 | 0 | 0 | 0 |
| | $R_o$ | 0.091 | 0.096 | 0.089 | 0.081 | 0 | 0 | 0 | 0 |
| | $t_d$ | 1.8 | 3 | 3 | 4.4 | 6.9 | 1245 | 1247 | 1248 |
| | $S_d$ | 24 | 29.7 | 28.7 | 46.6 | 21 | 42.1 | 39.6 | 38.3 |
| | $J_d$ | 7 | 7.1 | 7 | 7 | 12.2 | 6.9 | 18.3 | 19.6 |
| | $J_{90d}$ | 8 | 8.1 | 8 | 8 | 8.8 | 7 | 21.2 | 35 |
| | $T_{av}$ | 8.8 | 13.9 | 16.7 | 17.5 | 8.8 | 8.8 | 8.8 | 8.8 |
| | $h_{av}$ | 2.6 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| OMNET++ | $P_{gen}$ | 2060837 | 3433429 | 4120063 | 4577176 | 2061032 | 3433665 | 4120134 | 4577404 |
| | $P_{rec}$ | 2060824 | 3433399 | 4120007 | 4577110 | 2061026 | 2425433 | 2428200 | 2429529 |
| | $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | $S_{to}$ | 501 | 834 | 1000 | 1112 | 500 | 834 | 1001 | 1112 |
| | $S_{co}$ | 490 | 816 | 980 | 1088 | 490 | 0 | 0 | 0 |
| | $S_o$ | 2.9 | 5.7 | 8.6 | 10.6 | 0 | 0 | 0 | 0 |
| | $R_o$ | 0.1 | 0.1 | 0.1 | 0.1 | 0 | 0 | 0 | 0 |
| | $t_d$ | 6.5 | 7.8 | 12.4 | 15 | 3 | 1216 | 1218 | 1219 |
| | $S_d$ | 20.8 | 20.8 | 20.8 | 20.8 | 20.8 | 2.2 | 0 | 0 |
| | $J_d$ | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| | $J_{90d}$ | 8.8 | 9 | 9 | 9 | 5 | 5 | 5 | 5 |
| | $T_{av}$ | 8.4 | 14 | 16.9 | 18.8 | 8.4 | 9.9 | 10 | 10 |
| | $h_{av}$ | 2.6 | 2.7 | 2.9 | 3 | 2 | 2 | 2 | 2 |

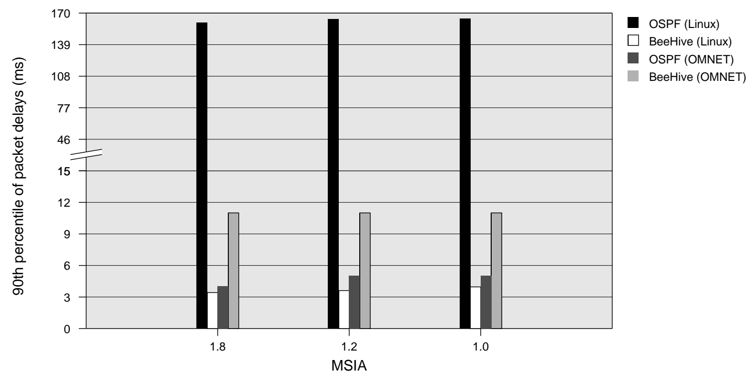
Table 5.4: Important performance values for Experiments 1 to 4 from

| Algorithm | | BeeHive | | | OSPF | | |
|---|---|---|---|---|---|---|---|
| Environment | MSIA | 1.8 | 1.2 | 1.0 | 1.8 | 1.2 | 1.0 |
| LINUX | $P_{gen}$ | 576914 | 862343 | 1035056 | 575898 | 861758 | 1035427 |
| | $P_{rec}$ | 576914 | 862343 | 1035056 | 575898 | 861758 | 1035427 |
| | $P_d$ | 100 | 100 | 100 | 100 | 100 | 100 |
| | $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| | $S_{to}$ | 556 | 831 | 998 | 555 | 831 | 999 |
| | $S_{co}$ | 556 | 831 | 998 | 555 | 831 | 999 |
| | $S_c$ | 100 | 100 | 100 | 100 | 100 | 100 |
| | $S_o$ | 0.738 | 2.354 | 3.026 | 0 | 0 | 0 |
| | $R_o$ | 0.014 | 0.011 | 0.109 | 0.003 | 0.002 | 0.002 |
| | $t_d$ | 2.4 | 2.5 | 2.9 | 160 | 164 | 164 |
| | $S_d$ | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 | 7.3 |
| | $J_d$ | 6.8 | 7 | 7 | 7 | 7 | 7 |
| | $J_{90d}$ | 8.9 | 8.9 | 8.9 | 7 | 7 | 7 |
| | $T_{av}$ | 4.7 | 7 | 8.5 | 4.7 | 7 | 8.5 |
| | $h_{av}$ | 2.281 | 2.6 | 2.642 | 2 | 2 | 2 |
| OMNET++ | $P_{gen}$ | 577368 | 865758 | 1038789 | 577343 | 865777 | 1038732 |
| | $P_{rec}$ | 577363 | 865751 | 1038781 | 577341 | 865773 | 1038727 |
| | $P_d$ | 99.999 | 99.999 | 99.999 | 99.999 | 99.999 | 99.999 |
| | $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| | $S_{to}$ | 556 | 834 | 1000 | 556 | 834 | 1000 |
| | $S_{co}$ | 553 | 829 | 995 | 553 | 829 | 995 |
| | $S_c$ | 99.5 | 99.5 | 99.5 | 99.5 | 99.5 | 99.5 |
| | $S_o$ | 1.636 | 2.469 | 2.981 | 0 | 0 | 0 |
| | $R_o$ | 0.031 | 0.036 | 0.039 | 0 | 0 | 0 |
| | $t_d$ | 7.5 | 7.6 | 7.6 | 4 | 4 | 4 |
| | $S_d$ | 5.2 | 5.2 | 5.2 | 5.2 | 5.2 | 5.2 |
| | $J_d$ | 6 | 6.6 | 6.6 | 5 | 5 | 5 |
| | $J_{90d}$ | 9.6 | 9.5 | 9.5 | 5 | 5 | 5 |
| | $T_{av}$ | 4.7 | 7.1 | 8.5 | 4.7 | 7.1 | 8.5 |
| | $h_{av}$ | 2.6 | 2.6 | 2.6 | 2 | 2 | 2 |

Table 5.5: Important performance values for Experiments 5 to 7

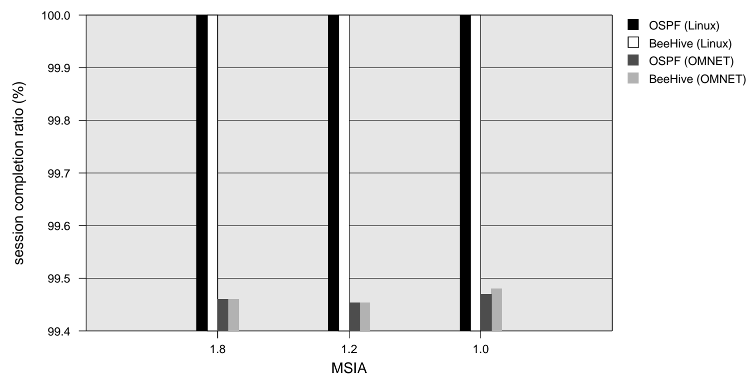



(a) Packet delivery ratio

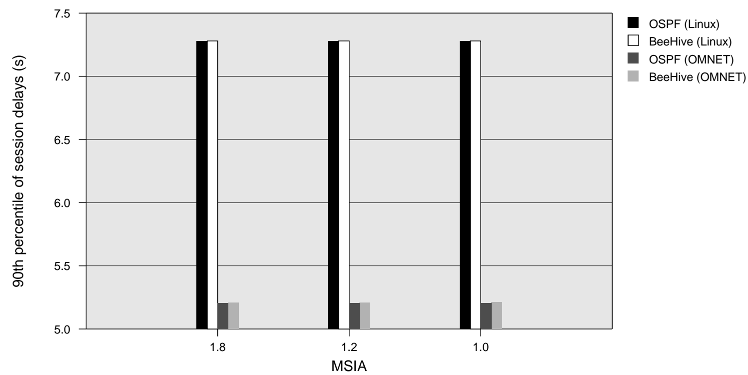



(b) 90th percentile of packet delays

Figure 5.15: Experiments 5-7 (packet delivery ratio and packet delay)




(a) Session completion ratio

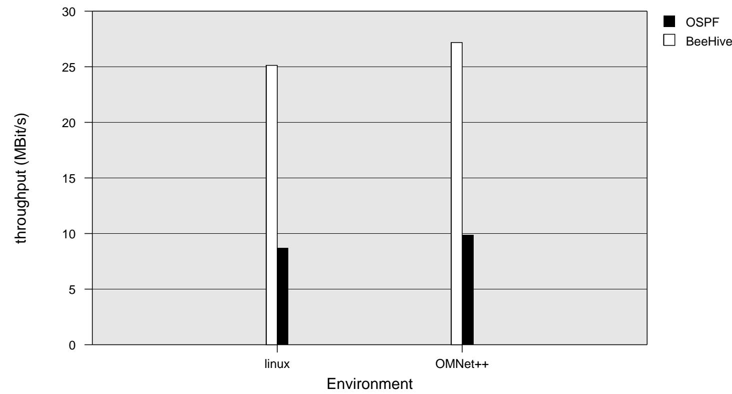



(b) 90th percentile of session delays

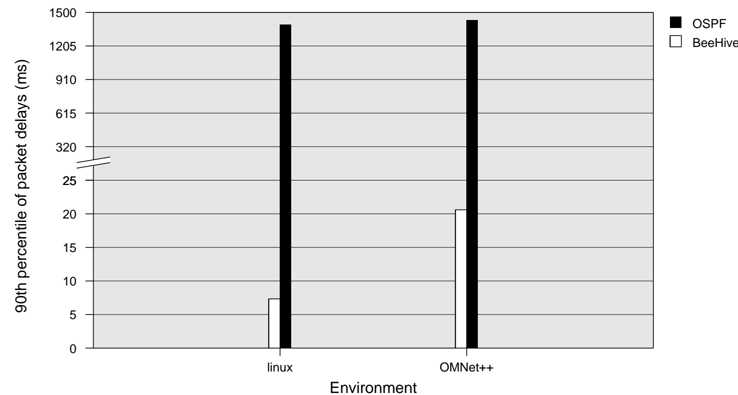
Figure 5.16: Experiments 5-7 (session completion ratio and session delay)

**Experiment 8**

We designed this experiment to unveil the true performance of *BeeHive*. We selected Etzel as the source node and Hagen as the destination node. Three distinct paths exist between Etzel and Hagen: Etzel-Hildebrand-Hagen, Etzel-Kriemhild-Siegfried-Hagen and Etzel-Gieselher-Gernot-Brunhilde-Hagen (see Figure 5.12(b)). We theoretically expect that *BeeHive* should be able to deliver approximately three times more packets at their destination than *OSPF*. The important parameters for the traffic generator are listed in Table 5.3. Figure 5.17(a) empirically substantiates our expectations: the packet delivery ratio of *BeeHive* is approximately 99.9% as compared to 35% of *OSPF*. Figure 5.17(b) shows that the packet delay of *BeeHive* is 20 msec as compared to 1431 msec of *OSPF*. The other important performance values are collected in Table 5.6. As expected, the throughput of *BeeHive* is approximately 24 Mbits/sec which is three times higher as compared to 8 Mbits/sec of *OSPF*. At this load, *OSPF* is unable to complete even a single session (Table 5.6) as compared to more than 90% of *BeeHive*. The session delay of *BeeHive* is significantly higher in a real network topology as compared with OMNeT++ because of the reasons already discussed. During a few runs *OSPF* was able to finish one or two sessions, and the session delay was approximately 60 seconds. Please note that the jitter of *BeeHive* is significantly smaller than of *OSPF* in the real network topology. The performance values of the algorithms are easily traceable from simulation into the real network of Linux routers.




(a) Throughput




(b) 90th percentile of packet delays

Figure 5.17: Experiment 8 (throughput and packet delay)

| Environment | Linux | | OMNET++ | |
|---|---|---|---|---|
| Algorithm | BeeHive | OSPF | BeeHive | OSPF |
| $P_{gen}$ | 3077866 | 3049156 | 3311173 | 3311178 |
| $P_{rec}$ | 3073359 | 1067326 | 3311071 | 1204964 |
| $P_d$ | 99.853 | 35.003 | 99.998 | 36.39 |
| $P_{loop}$ | 0 | 0 | 0 | 0 |
| $S_{to}$ | 433 | 433 | 435 | 435 |
| $S_{co}$ | 432 | 0 | 401 | 0 |
| $S_c$ | 99.8 | 0 | 92.82 | 0 |
| $S_o$ | 13.994 | 0 | 14.243 | 0 |
| $R_o$ | 0.224 | 0 | 0.151 | 0 |
| $t_d$ | 7.2 | 1348 | 13.7 | 1203 |
| $S_d$ | 50.4 | 60.3 | 39.6 | 0 |
| $S_{90d}$ | 54.4 | 67.7 | 39.6 | 0 |
| $J_d$ | 4.8 | 24.4 | 5 | 5 |
| $J_{90d}$ | 12.8 | 77 | 7.3 | 5 |
| $h_{av}$ | 3 | 2 | 2.9 | 2 |

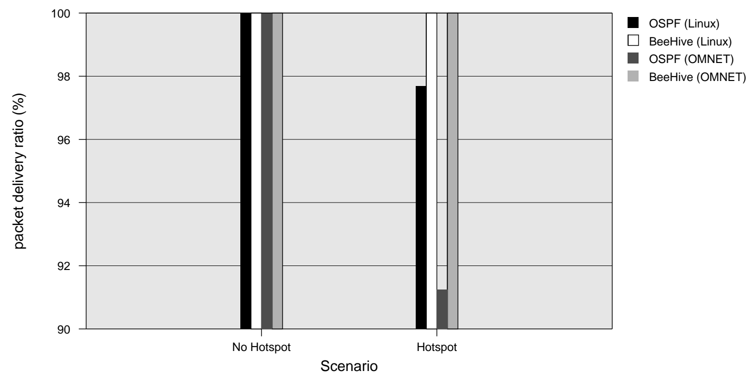
Table 5.6: Important performance values for Experiment 8

| Environment | Linux | | | | OMNET++ | | | |
|---|---|---|---|---|---|---|---|---|
| Algorithm | BeeHive | | OSPF | | BeeHive | | OSPF | |
| Scenario | No Hotspot | Hotspot | No Hotspot | Hotspot | No Hotspot | Hotspot | No Hotspot | Hotspot |
| $P_{gen}$ | 2134388 | 2367310 | 2133851 | 2202463 | 2145269 | 2845271 | 2145479 | 2845395 |
| $P_{rec}$ | 2134388 | 2367310 | 2133851 | 2151464 | 2145256 | 2845257 | 2145472 | 2596391 |
| $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $S_{to}$ | 868 | 869 | 868 | 868 | 870 | 870 | 870 | 870 |
| $S_{co}$ | 868 | 869 | 868 | 355 | 859 | 859 | 859 | 552 |
| $S_o$ | 2.578 | 4.199 | 0 | 0 | 3.403 | 4.769 | 0 | 0.797 |
| $R_o$ | 0.116 | 0.111 | 0.001 | 0 | 0.062 | 0.075 | 0 | 0 |
| $t_d$ | 4.3 | 3.6 | 1288 | 1246 | 6.575 | 5.8 | 3 | 409 |
| $S_d$ | 17.2 | 18 | 17.2 | 28.2 | 12.4 | 12.4 | 12.4 | 12.4 |
| $J_d$ | 7 | 6.6 | 7 | 11.4 | 5 | 1.9 | 5 | 1 |
| $J_{90d}$ | 8.1 | 8 | 7 | 18.8 | 7.3 | 3 | 5 | 1 |
| $T_{av}$ | 8.7 | 9.7 | 8.7 | 8.8 | 8.8 | 11.7 | 8.8 | 10.6 |
| $h_{av}$ | 2.8 | 2.8 | 2 | 2 | 2.7 | 2.4 | 2 | 1.7 |

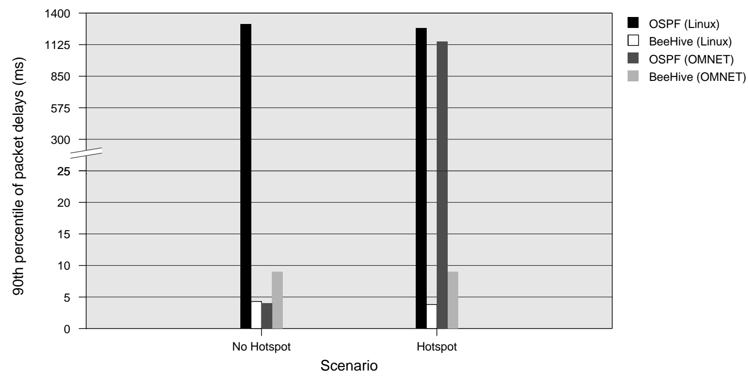
Table 5.7: Important performance values for hot spot experiments

**Experiment 9: Hot spot**

The purpose of this experiment is to study the behavior of the algorithms under hot spot traffic. A session-less hot spot traffic with MPIA = 0.001 sec from Etzel to Hagen was superimposed on the normal session-oriented traffic. The parameter values for session-oriented traffic are shown in Table 5.3. The hot spot remained active from 100 seconds to 800 seconds. Figure 5.18(a) clearly shows that the hot spot traffic is successfully delivered by *BeeHive*, both in simulation and real network. In contrast, the packet delivery ratio under *OSPF* decreased by 3% and 6%, in simulation and real network respectively. The packet delay of *BeeHive*, as shown in Figure 5.18(b), is significantly smaller than under *OSPF*. Similarly, Figure 5.19(a) shows that the hot spot has significantly effected the ability of *OSPF* to successfully complete the sessions while on *BeeHive* it has a negligible effect. The hot spot traffic has significantly degraded the session delay of *OSPF* in the real Linux network and it has negligible effect on *BeeHive*. The hot spot has helped in reducing the jitter (see Table 5.7) because now more packets flow between each (source, destination) pair, and this reduces the inter-arrival time of the packets, originating at the same node, at their destination node. One can easily see a significant difference in the number of packets generated during hot spot traffic, between OMNeT++ and Linux in Table 5.7. The reason is again due to the non-availability of high resolution timers in Linux, as described in the previous subsections. This explains the negligible effect of the hot spot traffic on the jitter values in the real network topology. The other important parameters are collected in Table 5.7.

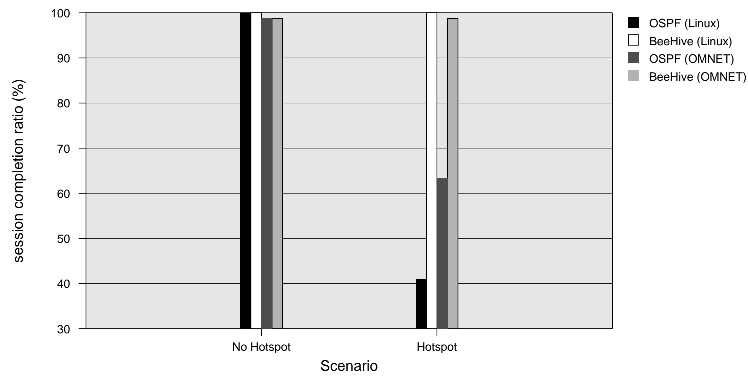



(a) Packet delivery ratio

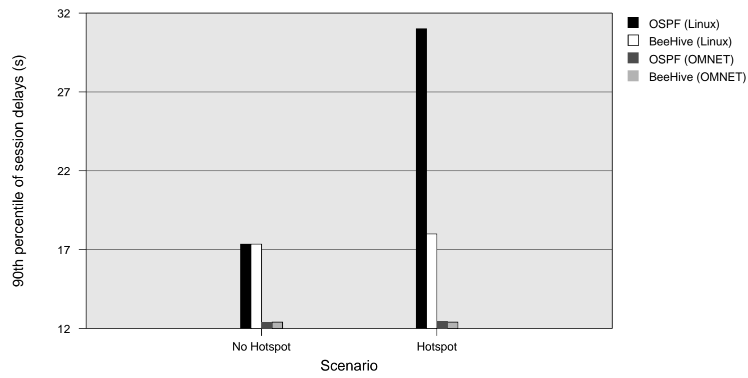



(b) 90th percentile of packet delays

Figure 5.18: Hot spot experiments (packet delivery ratio and packet delay)




(a) Session completion ratio

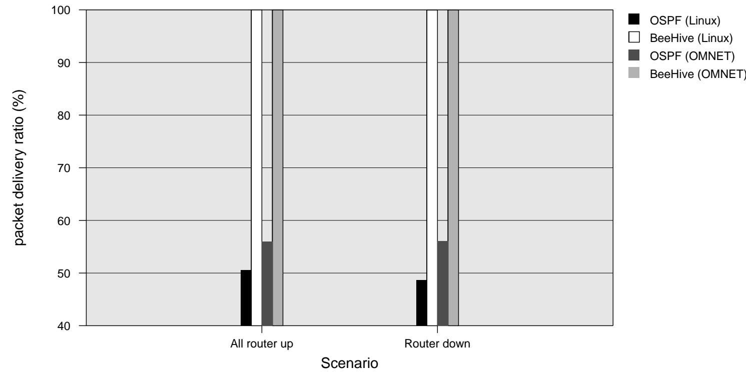



(b) 90th percentile of session delays

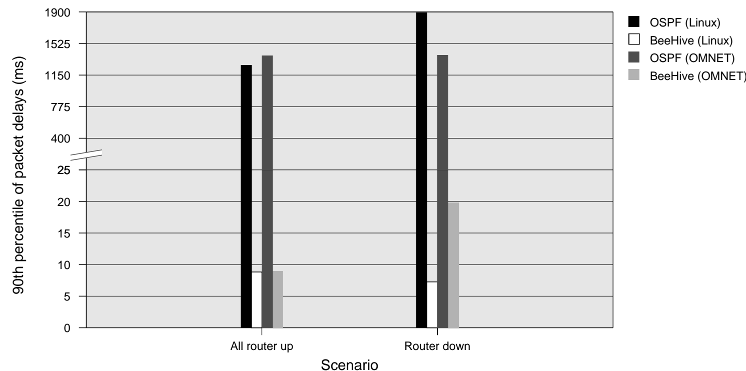
Figure 5.19: Hot spot experiments (session completion ratio and session delay)

**Experiment 10: Router down**

The purpose of this experiment was to study the fault tolerant behavior of *BeeHive* as compared with *OSPF*. A good routing algorithm should be able to quickly adapt its routes if a router crashes. The input parameters for this experiment are in Table 5.3. In this experiment Hildebrand crashed from 300 seconds to 600 seconds. We also repeated the experiments with the same traffic pattern but without any router crash. Figure 5.20(a) shows that the packet delivery ratio of *BeeHive* is significantly higher than of *OSPF* even when Hildebrand crashed. Similarly, Figure 5.20(b) shows that the packet delay of *BeeHive* is significantly smaller (15 msec) as compared with *OSPF* (1800 msec). The crashing of Hildebrand also has a negligible effect both on the session completion ratio (see Figure 5.21(a)) and the session delay (see Figure 5.21(b)) of *BeeHive*. The important parameters are collected in Table 5.8. *BeeHive* was able to quickly reroute the network traffic over two alternate paths: Etzel-Kriemhild-Siegfried-Hagen and Etzel-Gieselher-Gernot-Brunhilde-Hagen. One can see in Table 5.8 that the crashing of Hildebrand did not have any significant effect on the performance values of *BeeHive*. Please note that the results from the experiments suggest that *OSPF* is also able to react to the changes in topology but due to its single-path routing policy its performance values are significantly inferior to *BeeHive*. However, if we just look at *OSPF* in isolation then its packet delay significantly degraded once Hildebrand cashed (see Figure 5.20(b)). However, *OSPF* has approximately same packet delivery ratio in both cases (see Figure 5.20(a)).

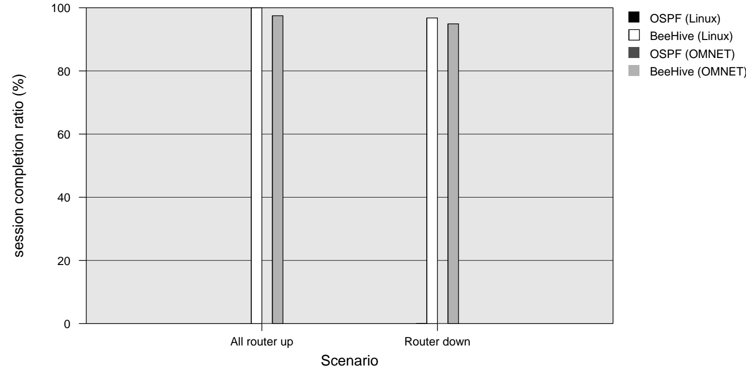



(a) Packet delivery ratio

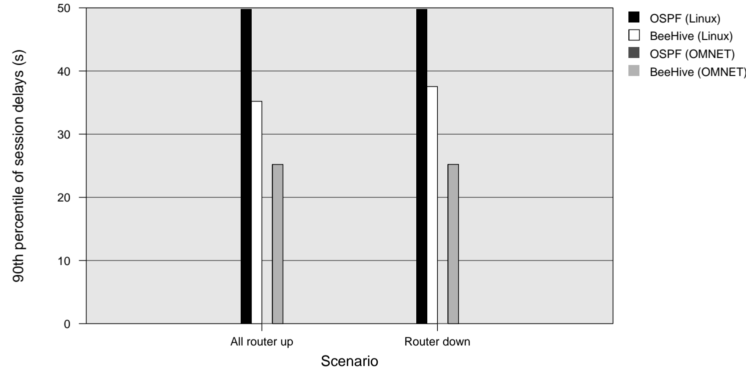



(b) 90th percentile of packet delays

Figure 5.20: Router down (packet delivery ratio and packet delay)




(a) Session completion ratio




(b) 90th percentile of session delays

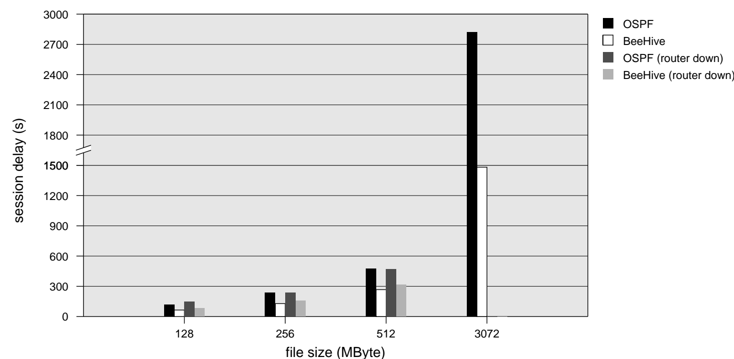
Figure 5.21: Router down (session completion ratio and session delay)

| Environment | Linux | | | | OMNET++ | | | |
|---|---|---|---|---|---|---|---|---|
| Algorithm | BeeHive | | OSPF | | BeeHive | | OSPF | |
| Scenario | No fault | Router down | No fault | Router down | No fault | Router down | No fault | Router down |
| $P_{gen}$ | 4289765 | 42792034 | 4260629 | 4275420 | 4329813 | 4330172 | 4330022 | 4330022 |
| $P_{rec}$ | 4289765 | 42780121 | 2153291 | 2082480 | 4329781 | 4329640 | 2426096 | 2429090 |
| $P_{loop}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $S_{to}$ | 866 | 866 | 867 | 911 | 870 | 870 | 870 | 870 |
| $S_{co}$ | 866 | 838 | 0 | 1 | 848 | 826 | 0 | 0 |
| $S_o$ | 2.743 | 19.98 | 0 | 0.24 | 7.246 | 9.687 | 0 | 1.675 |
| $R_o$ | 0.08 | 0.085 | 0 | 0 | 0.105 | 0.095 | 0 | 0 |
| $t_d$ | 7 | 7 | 1249.2 | 1364.6 | 7 | 10.1 | 1217 | 1214 |
| $S_d$ | 35 | 35.1 | 47 | 43 | 25.2 | 25.2 | 0 | 0 |
| $J_d$ | 6.4 | 4.3 | 18.9 | 18.8 | 5 | 5 | 5 | 5 |
| $J_{90d}$ | 10.1 | 7.4 | 28.1 | 28.1 | 7.3 | 7.3 | 5 | 5 |
| $T_{av}$ | 8.5 | 17.6 | 8.8 | 8.5 | 17.7 | 17.8 | 9.9 | 10 |
| $h_{av}$ | 2.3 | 3.1 | 2 | 2.3 | 2.7 | 3 | 2 | 2.3 |

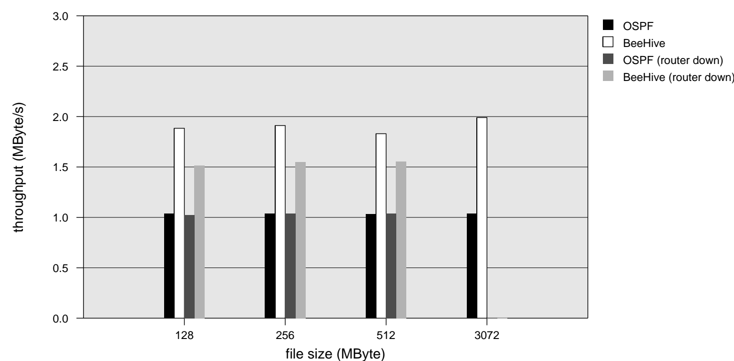
Table 5.8: Important performance values for router down experiments

### 5.6.2 Real world applications traffic engineering

The purpose of these experiments are twofold: one, to repudiate a strong thesis held by the networking community: *a stochastic routing algorithm brings subsequent packets out of order at their destination which then might confuse the TCP protocol as it expects an in-order delivery of packets*. Secondly, we wanted to demonstrate the benefits of the *BeeHive* protocol in real world applications. We installed an FTP server at Hagen, and files of different size were transfered from Etzel to Hagen using an FTP client. Please remember that FTP utilizes a reliable TCP protocol at the transport layer. We repeated the download process of each file ten times, and all the reported performance values are an average of the values obtained from the ten independent runs. We consider the results from these experiments crucial for convincing the networking community about the tangible benefits that Nature inspired algorithms like *BeeHive* might be able to provide. We conducted two sets of experiments: first, we started a download of a file of a particular size, second, we started downloading files, each of same size, after every minute, till 15 minutes.




(a) Session delay of FTP transfers




(b) Throughput of FTP transfers

Figure 5.22: FTP experiments

#### FTP: Experiment 1,2,3,4

In these experiments we downloaded four files of different sizes one after the other. The sizes of the files were 128 Mbytes, 256 Mbytes, 512 Mbytes and 3 Gigabytes. In another variation of the experiments Hildebrand crashed during the transfer. We wanted to investigate whether the algorithms could quickly react to the changes in the topology and to reroute the traffic on the other paths. The files were download from Etzel to Hagen. Figure 5.22(a) shows the time it took (in seconds) to transfer these files. Two observations are obvious from Figure 5.22(a): *BeeHive*

requires approximately half of the download time as compared with *OSPF*, in all scenarios. Also, the crashing of Hildebrand during the transfer did not significantly degrade the performance of the algorithms. The reason for the significantly smaller delay is that *BeeHive* is able to maintain higher throughput than *OSPF* (see Figure 5.22(b)). The results of the experiments are instrumental because they suggest that *BeeHive* is able to successfully transport TCP traffic and even with this protocol its performance is significantly better than under *OSPF*. We believe that the results will help in establishing the suitability of Nature inspired routing algorithms in real world networks.

**FTP: Experiment 5**

In this experiment we started downloading files of 128 Mbytes each from Etzel to Hagen every minute till 15 minutes. We wanted to investigate which algorithm has a smaller turn-around time and a smaller time to complete a single download. Figure 5.23 shows that the time to download each file in *BeeHive* is significantly smaller as compared with *OSPF*. The average time for each download, in case of *BeeHive*, is approximately 1000 seconds less as compared with *OSPF*. *OSPF* took 3642 seconds to finish the download of all 15 sessions as compared to 2042 seconds taken by *BeeHive*. This experiment further backs the results of the experiments from the previous subsection: *BeeHive* is able to seamlessly work with TCP, and its performance values are also significantly better than that of *OSPF*.

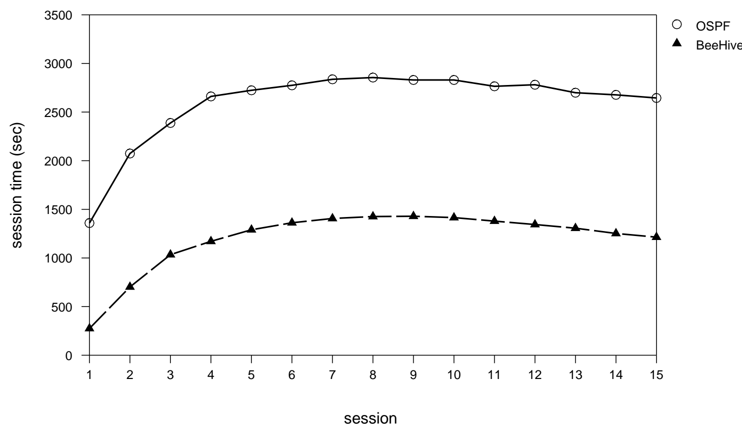



Figure 5.23: FTP experiments with 15 downloads

### 5.6.3 Hybrid traffic engineering

The motivation of doing experiments with hybrid traffic engineering is twofold: first, to use a traffic generator that has been developed by a third party to factor out any bias generated either in the traffic patterns or in the performance evaluation by our SQTG; second, to statistically evaluate different performance values for multimedia applications like Voice over IP (VoIP). VoIP is expected to capture a significant share of voice traffic in telecommunication industry. D-ITG served our purpose. It can synthetically generate TCP, UDP and VoIP traffic and then evaluate important performance values at the destination using a performance evaluation module. We generated UDP traffic, to verify that the tendency observed in the experiments using our SQTG is traceable to the results obtained from the D-ITG experiments. We also did experiments with VoIP traffic only, and VoIP traffic coupled with high UDP traffic load.

| Traffic Generator | $T_{av}$ | $t_d$ | $J_d$ | $P_{gen}$ | $P_{rec}$ | $P_d$ |
|---|---|---|---|---|---|---|
| SQTG | 25.1 | 7.2 | 4.8 | 3077866 | 3073359 | 99.9 |
| ITG | 24.5 | 3.4 | 3.7 | 2991185 | 2985347 | 99.8 |

Table 5.9: Performance values from SQTG and D-ITG

### UDP traffic

In D-ITG, a user can set the rate of packet generation, the size of a packet and its source and destination nodes. We chose a packet size of 512 bytes, packet rate of 6000 packets/sec, and the packets are sent from Etzel to Hagen (see Figure 5.12(b)). This scenario is semantically similar to the traffic load of Experiment 8 discussed in the previous section. We will like to emphasize an important observation here: D-ITG is not an open-loop traffic generator, rather it tries to generate only the number of packets which a routing protocol can easily transport without significant packet drops. One has to carefully interpret the results obtained from D-ITG because the packet delivery ratio of the compared algorithms might be the same but then the number of packets generated and delivered are usually different. Consequently, the throughput values would be significantly different. This behavior of D-ITG makes it difficult to systematically interpret and compare the performance values of different algorithms because the traffic patterns are adapted in response to the behavior of an algorithm.
The important performance values are collected in Table 5.10. Please note that the throughput of *BeeHive* (24 Mbits/sec) is approximately three times higher than that of *OSPF* (9 Mbits/sec). The average delay of *BeeHive* is 34 msec as compared to 71 msec of *OSPF*. However, we should remember that *BeeHive* is handling three times more packets as compared with *OSPF* in this scenario. The jitter value of *OSPF*, as expected, is better than *BeeHive*. Finally, we have collected important performance values for *BeeHive*, obtained from the current experiment and the ones from Experiment 8 of our SQTG in Table 5.9. In Table 5.9, one can easily correlate different performance values obtained from two different traffic generators. However, we can not make such a comparison for *OSPF* because D-ITG iteratively reduces the network load if an algorithm is unable to cope with it.

### VoIP traffic

The purpose of these experiments is to verify that *BeeHive* is able to handle multimedia traffic in a similar or a better fashion than *OSPF*. We started 10 parallel VoIP sessions between Etzel and Hagen which utilized G.711 [2] codec and with one sample per packet. One VoIP session requires approximately 64 Kbits/sec bandwidth. Ten sessions will need 640 Kbits/sec which is significantly less than the 10 Mbits/sec bandwidth available in simpleNet. The performance values are collected in Table 5.10. As expected, both algorithms are able to deliver approximately the same number of packets, and maintain the same throughput, packet delay and jitter values. We could not simulate large number of VoIP sessions because D-ITG simply can not manage a large network load. We expect that *BeeHive* can outperform *OSPF* at high load.

### VoIP traffic + UDP traffic

The above-mentioned scenario for VoIP traffic generally does not exist in the Internet [70]. Most of the time voice traffic is multiplexed with data traffic. Some users are surfing, some are downloading large audio/video files, and in parallel some users are involved in audio or video conferencing. The purpose of this experiment was to create an environment in which the 10 VoIP sessions have to share the network with a UDP session generating 3000 packets/sec of 512 bytes size per packet. All sessions are between Etzel and Hagen. We have collected the relevant parameters in Table 5.10. *BeeHive* is able to deliver approximately 1 million more packets to their destination by maintaining a throughput of 13 Mbits/sec as compared to 8.5 Mbits/sec of *OSPF*. The packet delay of *BeeHive* is 2.6 msec as compared to 48 msec under *OSPF*. At this load, *BeeHive* is able to maintain a jitter value of 1 msec as compared to 4 msec of *OSPF*. One can easily conclude that in

| Scenario | Algorithm | $T_{av}$ | $t_d$ | $J_d$ | $P_{av}$ | $P_{gen}$ | $P_{rec}$ | $P_d$ |
|---|---|---|---|---|---|---|---|---|
| UDP | OSPF | 8.9 | 72 | 1.9 | 2170 | 2170526 | 2170526 | 100 |
| | BeeHive | 24.5 | 34.2 | 3.7 | 5981 | 5994045 | 5982370 | 99.8 |
| VoIP | OSPF | 0.7 | 0.2 | 0.012 | 995 | 1000000 | 995719 | 99.57 |
| | BeeHive | 0.7 | 0.4 | 0.2 | 995 | 1000000 | 996039 | 99.6 |
| VoIP& UDP | OSPF | 8.5 | 48.6 | 5 | 2899 | 2899329 | 2899158 | 99.9 |
| | BeeHive | 13 | 2.7 | 1 | 3991 | 3999980 | 3991418 | 99.79 |

Table 5.10: Important performance values for UDP and VoIP experiments

case of hybrid traffic patterns, *BeeHive* is able to outperform *OSPF* considering all performance values. We intend to modify D-ITG in such a manner that it can manage a higher number of VoIP sessions along with UDP and TCP traffic, to further validate our assumption that the true merit of *BeeHive* is visible at large network loads. However, the results from the current experiments can still be considered as a break-through for Nature inspired routing algorithms because for the first time, to our knowledge, a Nature inspired routing protocol that has been tested in a real topology of Linux routers showed throughout quantifiable benefits with real applications.

## 5.7 Summary

The major contribution of the work is a *Natural Engineering* approach that we followed during the development of our routing protocol *BeeHive* and realizing it inside the network stack of the Linux operating system. The engineering approach was instrumental in incorporating only those features in the *BeeHive* algorithm in the OMNeT++ simulator that are also available in the network stack of Linux. We also designed a protocol verification framework, in which we developed a comprehensive performance evaluation framework and implemented it both in OMNeT++ and a Linux network. The framework allowed us to generate similar traffic patterns, both in the simulation and in a real network of Linux machines. The basic assumption of the framework is: *if the simulation results of* BeeHive *are of any significance then its behavior should be traceable in a real network of Linux machines as well.* We designed a variety of experiments which utilized real-world applications like FTP and synthetic traffic generators like SQTG and D-ITG to demonstrate that the *behavior of BeeHive from simulation is consistent to its behavior in a real network of Linux routers.* The results obtained from simulation and a real network of Linux machines also show a strong correlation and in our view this is a breakthrough work in favor of Nature inspired routing algorithms. Because to our knowledge, it is the first study that has empirically refuted the suspicious notion held by the telecommunication industry that *Nature inspired algorithms are not realizable with the existing resources of software and hardware.* Of course, our findings will have to be further confirmed in large networks. Our current results are quite an intriguing stimulation for such plans.
We have deliberately used the low-end Pentium III based machines for the implementation of *BeeHive* to prove the fact that the algorithm is realizable with the existing infra-structure and yet its performance benefits are unequivocally noticeable in real world networks. Future work includes testing the algorithm on large scale topologies.

# 6

# Conclusion and Future Works

*This chapter provides a comprehensive conclusion by emphasizing the scientific and technical contributions of our project* BeeHive*. We then outline our vision of extensive future research that could follow the successful conclusion of our work. We put special emphasis on an intelligent and knowledgeable router, Nature Inspired Decentralized and Autonomous Router (NIDAR), which can become a state-of-the-art router for networks of the new millennium. Finally, we suggest that the time has come to start a* Natural Engineering *program in our universities in order to successfully translate novel and cost effective Nature/Bio inspired business solutions for highly competitive markets.*

## 6.1 Conclusion

We developed a a dynamic, simple, efficient, robust, flexible and scalable multi-path routing algorithm, *BeeHive*, for packet switched fixed telecommunication networks. The algorithm is a first important step in endowing the network layer with intelligence and knowledge. An intelligent network layer is able to optimally manage its network resources. In this work, our focus was to provide a solution to the challenges in *traffic engineering* by designing a multi-path routing algorithm for IP networks. Such a paradigm will try to exploit the network bandwidth by utilizing the connectionless feature of IP. This is in contrast to existing approaches of either managing virtual circuits on top of the IP layer (MPLS) or administering the resources on a per flow basis (RSVP). These approaches try to solve the QoS guarantee problems for streaming multimedia applications while utilizing *OSPF* at the IP layer. Our hypothesis is that simple and cost-effective solutions are possible for QoS routing if we reengineer the networking layer and the routing protocols.
An ideal or dream network layer should have the following features: *it should be able to discover and manage its routes in a decentralized and asynchronous fashion without the need to have access to the global view of the network topology. The diagnosis of fault and its remedial management should be embedded in the routing system. The network layer should be able to guarantee its resources to streaming network traffic and should be able to enhance the network performance for best effort traffic. It should be able to do multi-objective optimization between competing and conflicting demands according to the network state.*
Our observation is that different Natural colony systems, for example a honey bee colony, are able to achieve similar tasks through simple individuals that have limited memory and processing abilities. However, they follow simple rules to coordinate their activities. As a result, an intelligent and coherent pattern emerges at a colony level which is beyond the capabilities of any individual. According to Seeley [170], *a honey bee colony can thoroughly monitor a vast region around the hive for rich food sources, nimbly redistribute its foragers within an afternoon, fine-tune its nectar processing to match its nectar collecting, effect cross inhibition between different forager groups to boost its response differential between food sources, precisely regulate its pollen intake in relation to its ratio of internal supply and demand, and limit the expensive process of comb building to*

*times of critical need for additional storage space.* These observations motivated us to make the following hypotheses in the beginning of our work, which we reproduce for the sake of clarity:

(a) **H1:** If a honey bee colony is able to adapt to countless changes inside the hive or outside in the environment through simple individuals without any central control, then an agent system based on similar principles should be able to adapt itself to an ever changing network environment in a decentralized fashion with the help of simple agents who rely only on local information. This system should be dynamic, simple but efficient, robust, flexible, reliable and scalable because its natural counterpart has got all of these features.

(b) **H2:** If designed with a careful engineering vision, Nature inspired solutions are simple enough to be installed on real world systems. Therefore, their benefit-to-cost ratio should be better as compared with existing real world solutions.

In Chapter 3, we developed an agent model that is inspired from the communication and evaluative features of a honey bee colony. The *bee agents* in our model have a simple behavior. As a result, the algorithm is able to take routing decisions in a decentralized and asynchronous fashion. We have conducted extensive simulations in OMNeT++ simulator to show the advantages of our algorithm over existing state-of-the-art routing algorithms developed by Nature inspired routing community. The *BeeHive* algorithm is able to achieve better performance values with a simple agent model. We were able to collect a comprehensive set of performance values through our performance evaluation framework to study the behavior of a routing protocol over a vast operational landscape.
We then defined a power metric for a routing protocol, which models the performance of a routing algorithm based on a number of parameters. We also defined a productivity metric that shows the benefit-to-cost ratio of a routing protocol in an unbiased manner. We then developed a scalability model to study the scalability of a routing protocol by conducting extensive experiments on six topologies varying in their size and complexity. We concluded that *BeeHive* is scalable for most of the network configurations. The results discussed in Chapter 3 and Chapter 4 are significant to confirm the validity of H1.
We then moved to a cardinal step in our *Natural Engineering* approach, in which we developed an *engineering model* from the simulation model and implemented it in the network stack of the Linux kernel. We developed the same traffic generator both in simulation and the application layer of network stack of the Linux kernel. We then tested our algorithm in a real network of Linux routers and compared the performance values of our algorithm, both in simulation and in real network, with *OSPF*. We can conclude from the results of the extensive experiments in Chapter 5 that the performance values of *BeeHive* were traceable from simulated to real networks. *BeeHive* is the first Nature inspired routing algorithm, according to our knowledge, that has been implemented and tested in real networks with the help of existing resources. The success of our efforts lie in an engineering approach that we followed in our protocol development cycle. The success in this phase also validates H2.
*BeeHive* discovers and evaluates multiple-paths in a deterministic fashion by utilizing a variant of breadth first search. It does not discover all possible multiple paths. Rather only those multiple paths are utilized whose quality is above a certain threshold. However, *BeeHive* then spreads the data packets on multiple paths in a stochastic fashion in order to achieve better performance values. Our experience suggests that *BeeHive* properly combines the deterministic elements with the stochastic elements in a routing algorithm. Therefore, the proper classification of *BeeHive* is shown in Figure 6.1.

**Concluding remark:** We maintain the implicit assumption throughout our work that the resources of a network including bandwidth have to be utilized in an efficient manner. The existing fiber optic networks of terabytes capacities provide a substantially greater bandwidth per user as compared to the networks that existed only a couple of years ago. This might obviate the need for multiple-path routing algorithms in the short term. However, the lesson of history is that whenever a resource is in abundance, new and powerful applications are developed that properly utilize that resource [207]. We strongly believe, therefore, that the need for a network layer that utilizes its resources in an efficient manner through intelligent routing algorithms can

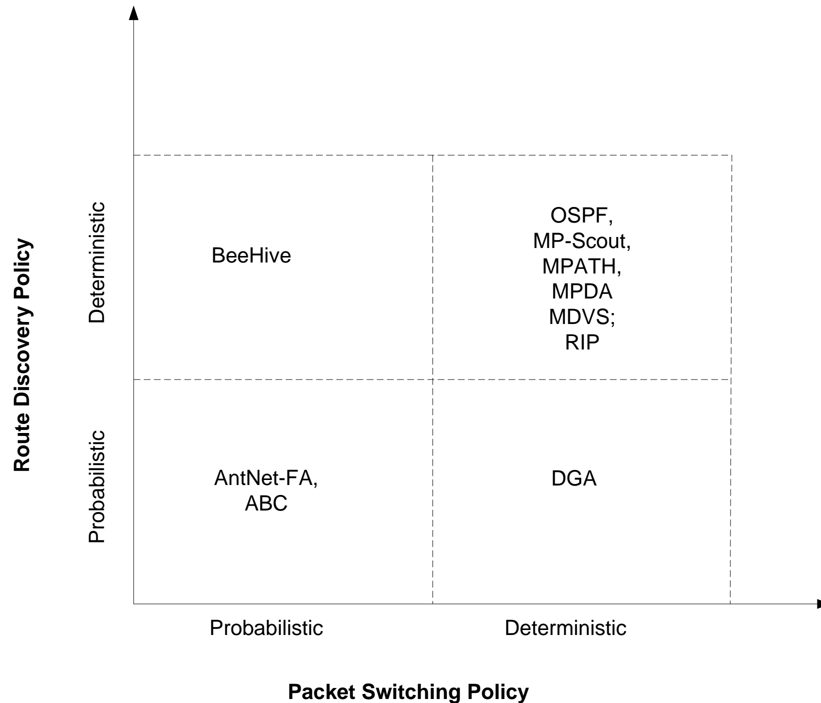



Figure 6.1: Routing classification

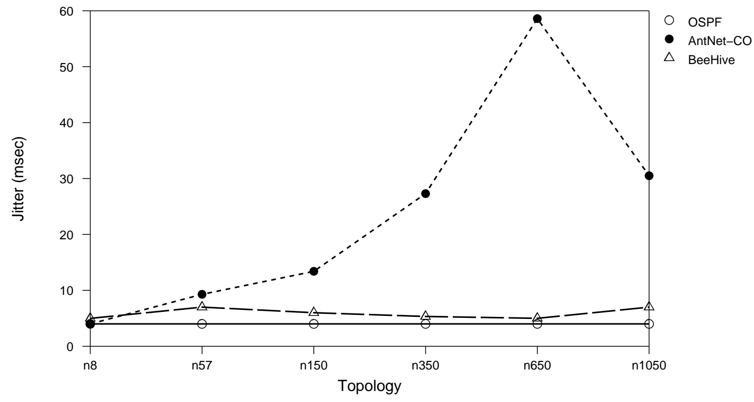
not be underestimated in the long term.

## 6.2 Future research

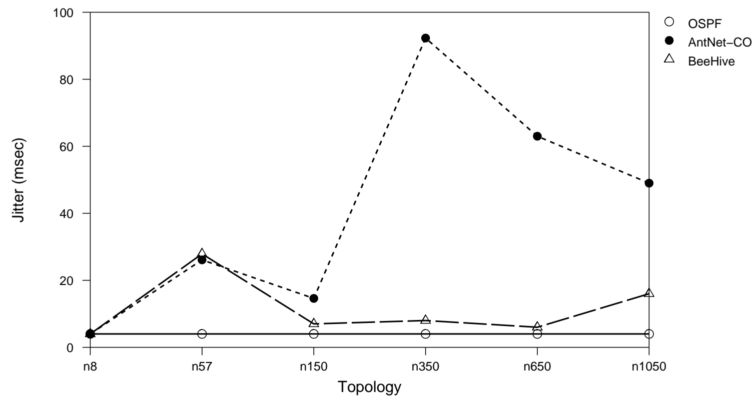
We believe that the objectives achieved in our project *BeeHive* are substantially encouraging, especially for Nature inspired algorithms community, because the work presents a simple Nature inspired Multi-agent system that is realized both in simulation and inside the network stack of the Linux operating system. After working for four years on this exciting project, we honestly believe that the accomplishments have to be followed with rigorous design and development efforts in order to translate the prototype to an intelligent real world routing system. We did not emphasize the related implementation details too much because we were interested in illustrating a "proof of concept" in the first phase of the project. We believe that the following issues need to be addressed in any future research work.

### 6.2.1 Quality of Service (QoS) routing

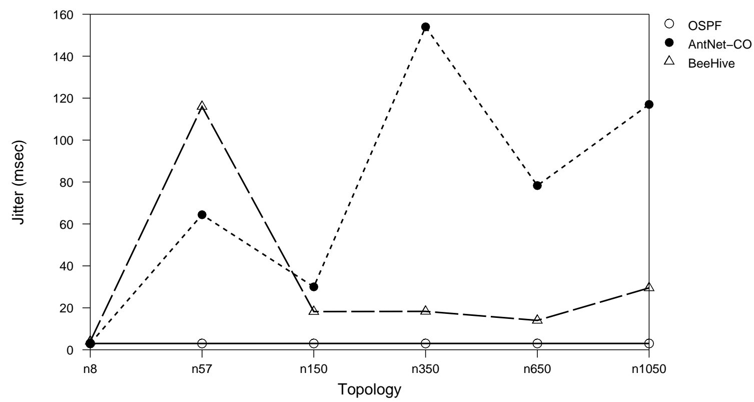
Our results in Figure 6.2 demonstrate that under high network traffic load the jitter value of *BeeHive* is significantly higher as compared with *OSPF*. The jitter values of *BeeHive* are comparable to *OSPF* at MSIA = 4.6 sec (see Figure 6.2(a)). The jitter values of *BeeHive* are also comparable with *OSPF* at MSIA = 2.6 sec (see Figure 6.2(b)) with the exception of n57. However, the jitter values of *BeeHive* are significantly inferior to *OSPF* at MSIA = 1.6 sec (see Figure 6.2(c)). This is naturally due to the stochastic spreading of data packets on multiple paths as per their quality (Please remember that *BeeHive* delivers significantly more packets with smaller delays at higher loads). Consequently, any two subsequent packets from the same session may follow different paths and this leads to a higher jitter value. Therefore, we believe that the future research has to




(a) MSIA = 4.6 sec




(b) MSIA = 2.6 sec




(c) MSIA = 1.6 sec

Figure 6.2: Jitter (msec)

address this issue. We propose three solutions to tackle this problem:

- *Time based routing stability* approach will ensure that the routes, once selected, remain valid for a certain time $t_{valid}$. In this way, the decision to stochastically route data packets does not happen at the arrival of each data packet. Rather it only happens at regular time intervals. We believe that by utilizing time based stability of routing decisions, we can still achieve a better jitter value without compromising the performance of the algorithm. The disadvantage is that the stability interval is independent of the network traffic load.

- *Load based routing stability* approach will ensure that the routes, once selected, remain valid for a certain number of data packets $n_w$. It means that the decision to stochastically route data packets is taken for every $n_w$ packet. The advantage of this approach is that the stability interval adapts itself according to the network traffic load: the stability interval will decrease with an increase in the traffic load and vice versa.

- *Priority routing* approach will treat the multi-media streaming packets in a priority manner, in which the routes for these types of traffic are reserved at the start of an application session. We will still be able to distribute the traffic on multiple paths but only on a session basis rather than on a packet basis for these types of traffic. However, the normal network traffic will be handled in a best effort manner.

We believe that an extensive study based on the above-mentioned three approaches will provide a working and optimum solution to the Quality of Service (QoS) routing even under high network traffic loads.

### 6.2.2 Cyclic paths

The data packets in the *BeeHive* algorithm can follow a cyclic path due to its stochastic routing property. The number of data packets that follow a cyclic path depend on the topology and the network traffic. Figure 6.3 shows the distribution of packets that followed cyclic paths in different topologies for different traffic patterns. We have given beneath each subfigure the topology name, the MSIA value and the percentage of packets that followed cyclic paths. $P(1)$ is the percentage of packets that followed a cyclic path once, $P(2)$ is the percentage of packets that followed the cyclic paths twice and $P(> 3)$ is the percentage of packets that followed the cyclic paths thrice or more. We can safely conclude from our results that the cyclic paths are only short lived and the probability that a packet might follow a cyclic path for the second time is approximately 1.0% (total probability) or below in almost all cases. The total probability of following a cyclic path second time in Figure 6.3(b) is 24% of 3.79% which is 0.91%.
The results of our experiments suggest that even with the cyclic paths, *BeeHive* is able to outperform *OSPF*. Nevertheless, if we can make *BeeHive* a loop free algorithm, then it will improve its acceptability, because Networking community puts a significant emphasis on loop free routing. However, the real challenge is *to achieve loop freedom in a stochastic routing algorithm without utilizing agents that do not have a stack memory.*

### 6.2.3 Formal analysis framework

The formal model of a routing protocol is an important element of *protocol engineering* [113]. In our current project, we could not emphasize on this point. The real challenge in the formal analysis is *to formally analyze the behavior of a time-varying, dynamic, non-linear self-organizing system.* To our knowledge, the formal treatment of such systems has received little attention. We need to develop new techniques for the formal analysis of such systems. The existing work treats the computer networks as "Stochastic Networks" [172] in which the behavior of a network is modeled in terms of the equilibrium probability distribution of packets on different routes. The equilibrium distribution is used to define objective functions or constraints to model performance values such as throughput and packet delay. Such probabilistic methods are also utilized to model

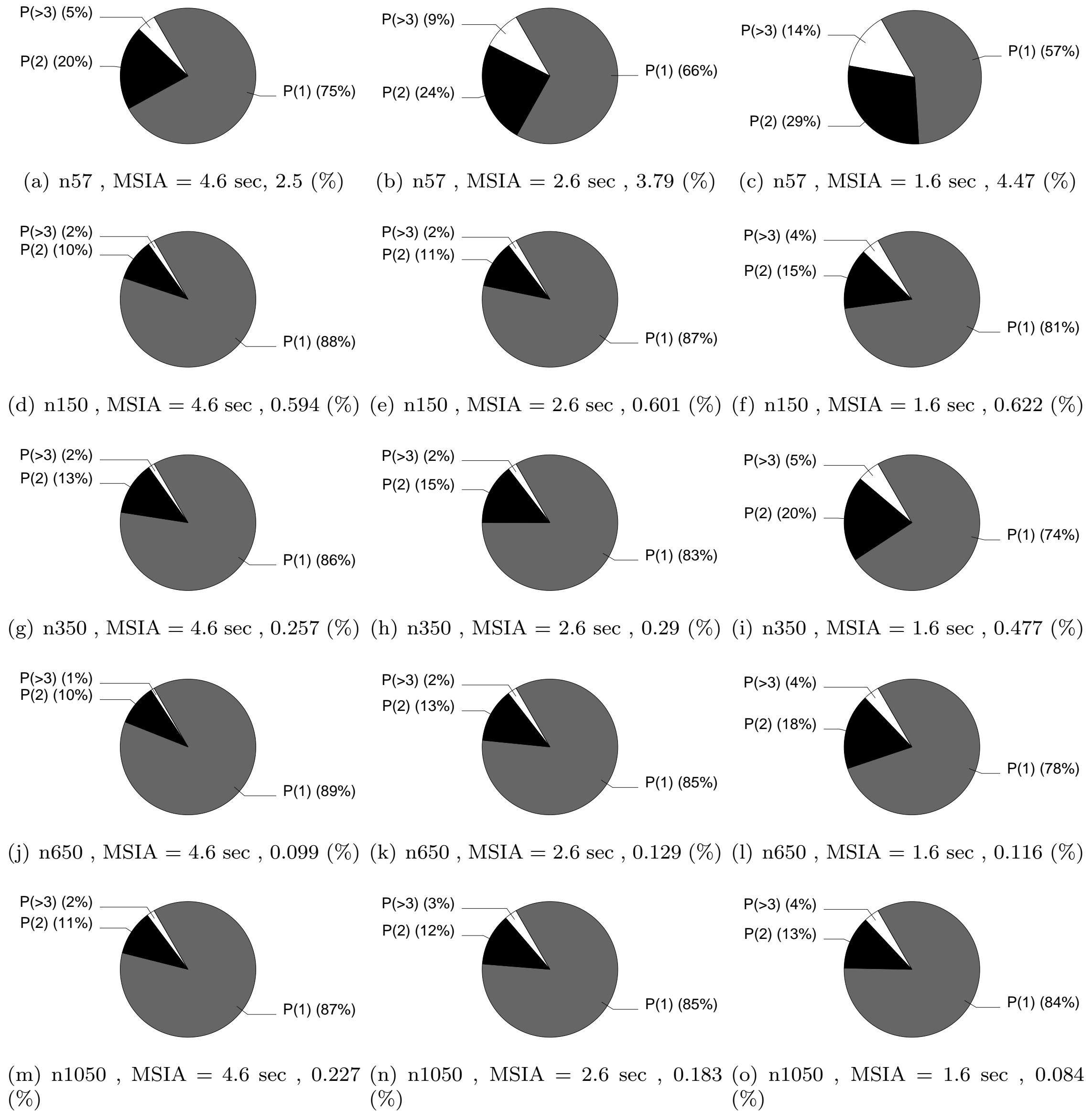



Figure 6.3: Distribution of packets that follow cyclic paths

the Internet and the Web [13]. In [44], an initial attempt has been made to analyze Nature inspired agent-based routing algorithms. However, the "Stochastic Networks" are unable to model the self-organizing and emerging behavior of Nature inspired routing algorithms. Recently, physicists have started applying the principle of "Statistical Physics" to come up with an empirical and theoretical research framework for analyzing the behavior of such self-organized network systems [151, 68]. We already mentioned in Chapter 3 that Sumpter introduced the Weighted Synchronous Calculus of Communicating Systems (WSCCS) to formally model an agent-based system for a honey bee colony. Using this model, he studied the foraging behavior of a honey bee colony [192], which of course is a self-organizing system.
We have done some initial work in [122] utilizing the "stochastic networks" approach to get an insight into the behavior of *BeeHive* in a very simple topology. A preliminary work has also been done in [220], in which we modeled the behavior of *bee agents* in our *BeeAdHoc* [217, 218] algorithm by utilizing the WSCCS. The initial results from the theoretical model are encouraging. We believe that a combination of these different approaches will help in developing a comprehensive formal framework for analyzing self-organizing network systems.

### 6.2.4 Security

We believe that a thorough investigation of security threats of Nature inspired routing algorithms is necessary. The developers of Nature inspired routing algorithms always hold the view that one can always trust a *routing agent*, which of course is not true in real networks. In [245] a preliminary treatment of security issues relating to *AntNet* has been presented. We have finished a comprehensive work on analyzing the security threats that may arise as a result of deploying our *BeeAdHoc* algorithm in a MANET environment [220]. We developed a novel security framework inspired by Artificial Immune Systems (AIS) to counter threats of *BeeAdHoc*.
The mobile agents in Nature inspired routing algorithms are launched periodically in short intervals of time. If we use the traditional methods that are based on cryptographic techniques and digital signatures then this will significantly increase the processing overhead. Our experience working with *BeeGuard* [220] suggests that such classic security approaches are not suitable for MANETs and we believe that the corollary can be easily applied to fixed networks as well. The real challenge in agent-based secure routing algorithm is: *to ensure security without utilizing existing cryptographic techniques and secure IP protocols like IPSec. This is to avoid the processing overhead in routing.* We believe that ensuring security with the above-mentioned constraints would be a challenging and exciting effort.

### 6.2.5 Intelligent and knowledgeable network engineering

We believe that this task can only be achieved by launching an interdisciplinary effort that requires cross fertilization among different areas of engineering and science, on one hand, and close cooperation between academia and industry on the other hand. The final aim of the research should be to design and develop a dedicated Nature Inspired Decentralized and Autonomous Router (NIDAR) that can seamlessly replace existing routers in the Internet. NIDAR is to be designed from scratch with *Intelligence and Knowledge* as the key principles. The conceptual block level diagram of NIDAR is shown in Figure 6.4. It shows the overwhelming complexity of achieving the task. Our vision is that the task can be achieved in an efficient and systematic fashion if we divide our objective into three important building blocks: hardware, software and performance evaluation.

#### Hardware

NIDAR is to be designed with an engineering vision that simplifies its installation in real networks. As the routers are required to perform packet switching in real time, it is not advisable to host any additional computational intensive tasks on such a system. This approach is in line with the recommendations made by Yu in his work [242]. Our suggestion is to develop NIDAR hardware

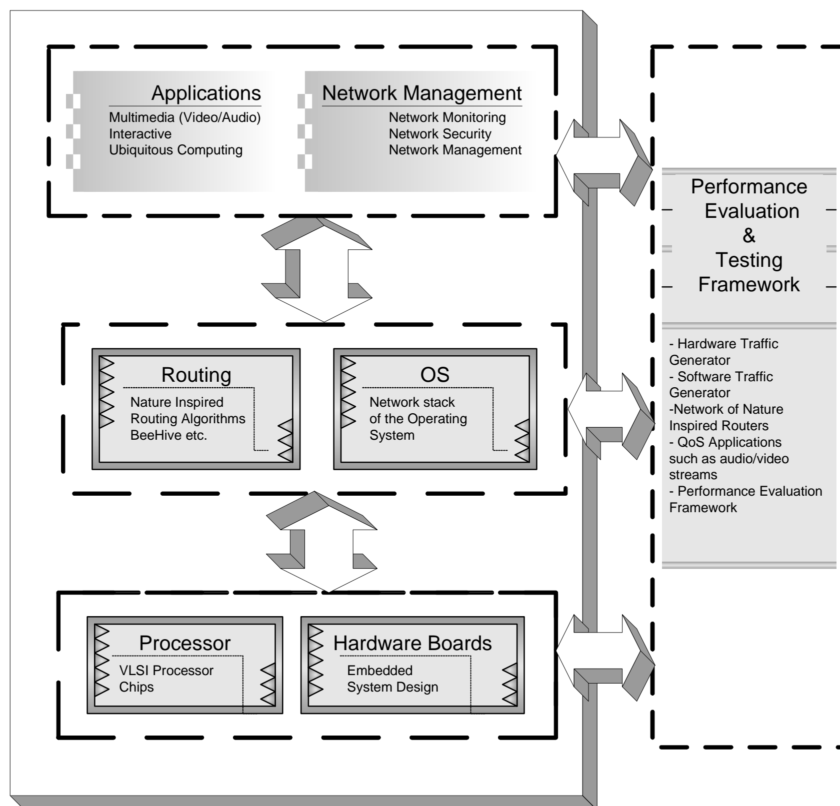



Figure 6.4: Nature inspired distributed and autonomous router

consisting of two sub-systems: one sub-system solely responsible for the packet switching, and one sub-system performing the agent processing tasks. We would like to realize the packet switching component of our algorithm in a chip in order to achieve maximum efficiency.
A very attractive option can be the possibility of remote and on-the-fly upgrading of firmware software on the agent processing sub-system. This feature will help in downloading the new updates of the algorithm and potential error fixes without bringing back the routers in service centers.

#### Software

The software component also consists of two subcomponents: system and application. The system component consists of an operating system for our hardware platform described in the previous subsection. We believe that a careful design of an operating system architecture is a rudiment step in realizing an intelligent network stack. In an intelligent network stack, the important information can be exchanged between different layers through a network object [42]. A network object defines an interface for each layer for storing or retrieving its information. In this way, each layer is able to acquire a comprehensive view of the network state in order to make intelligent decisions that could not be made based solely on its information. However, this cross-layer design must be achieved without violating the open system architecture. We believe, the real time multimedia audio/video applications, utilizing such an architecture, can interactively communicate with the network layer. Consequently, either they will get better Quality of Service (QoS) guarantees or they can gracefully adapt to the available network resources .
The other important benefit of utilizing agent-based routing systems is that *network management* can be easily incorporated in the network layer that involves network fault detection, route repair and recovery, resource management, traffic policing etc. Such an intelligent network system appears to become a core component of future networks.

#### Performance evaluation

The performance evaluation of NIDAR in hardware and software with the help of a network traffic generator will be an important validation step. We believe that the performance must be evaluated both in hardware and software to comprehensively analyze the true benefits of Nature inspired routing algorithms. The testing and evaluation should be done in real network environments with a set of different applications in order to continuously improve the design of NIDAR through feedback channels during an early stage of design and development. The outcome of this approach should report the quantitative gains over the existing routing systems. The evaluation and validation of a routing protocol is an important step of *protocol engineering* [113].

### 6.2.6 Bee colony metaheuristic

We should emphasize that the foraging model described in [171] [192] can be used for developing any multi-objective optimization problems for dynamic environments. The important concepts of *BeeHive*, such as the *bee agent propagation* algorithm and *bee agent communication* paradigm, can be applied to any optimization problem that can be represented in a graph. However, the real strength of *BeeHive* concept lies in dynamic environments in which resources have to be managed in an optimum fashion or in multi-objective optimization in which competing and, at times, conflicting requirements have to be incorporated in the objective function. We believe that it is possible to generalize the bee behavior to develop a new metaheuristic for optimization problems, that can be applied to different continuous and discrete optimization problems. We call it *Bee Hive Optimization (BHO).*

## 6.3 Natural Engineering: The need for a distinct discipline

We believe now is the suitable time that we seriously evaluate the feasibly of starting a distinct discipline of *Natural Engineering* in our universities. In this discipline, the novel algorithms and systems, which are inspired from Natural systems and which are developed by Natural computing community, can be engineered for real world applications. However, the major emphasis should be on cost effective design that can bring substantial saving in costs of designing, manufacturing, installing and maintaining such systems. We believe that such an engineering discipline requires a strong interdisciplinary effort resulting in revolutionary design and development paradigms, paving the way to the development of products that are impossible to achieve with classic engineering principles.

# A

# Software Protocol Engineering for Linux Routers

*This appendix describes a software engineering approach that will help the developers of the routing algorithms to support their algorithms inside the network stack of Linux operating system. We explain in detail the network infra-structure of Linux operating system and then identify the important interfaces for the implementation of novel Nature inspired routing algorithms. We believe, the appendix will act as an important guideline for the Networking community in general and Nature inspired routing community in particular, for realization and evaluation of their novel algorithms in real Linux routers.*

## A.1 Networking code

In [1]Figure A.1 one can see the hierarchy of the network code in the Linux kernel. The code that is relevant for networking is located in *net/*. For our work, IPv4 protocol is of most relevance. Its relevant files are contained in the folders *ipv4/* and *core/*. The corresponding header files can be found in the *include/* folder.

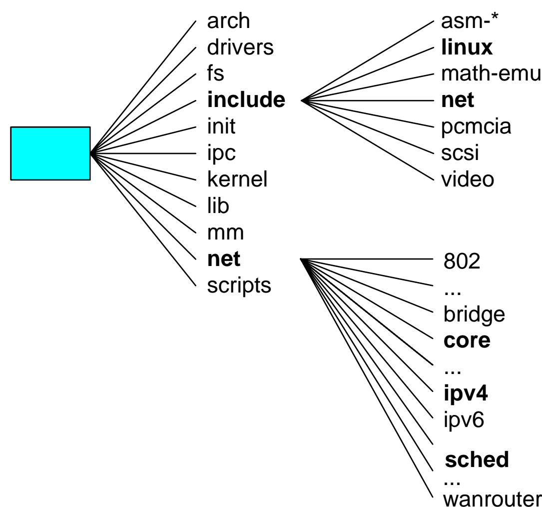



Figure A.1: Networking code in the Linux kernel tree [157]

[1]The appendix is also published as a research report TR-800. Horst F. Wedde, Muddassar Farooq and Alexander Harsch. *Software Protocol Engineering for Linux Routers.* Technical Report 800, School of Computer Science, University of Dortmund, 2005.

## A.2 Data structures

A packet is processed by multiple layers inside the network stack of the Linux kernel. The processing involves adding or removing the headers by protocols of different layers. The efficiency of these operations plays an important role in influencing the overall performance of the TCP/IP stack. Therefore, Linux kernel supports a special data structure *sk_buff*, which stands for socket buffer. It references the *net_device* structure and the *sock* structure which will be discussed later in the appendix.

### A.2.1 socket buffer sk_buff

The header of file in which the *sk_buff* structure is defined is **include/linux/skbuff.h**. Each packet processed by the TCP/IP stack is accessed through this structure.

```
struct sk_buff {
        /* These two members must be first. */
        struct sk_buff                  *next;
        struct sk_buff                  *prev;

        struct sk_buff_head             *list;
        struct sock                     *sk;
        struct timeval                  stamp;
        struct net_device               *dev;
        struct net_device               *real_dev;

        union {
        struct tcphdr                   *th;
        struct udphdr                   *uh;
        struct icmphdr                  *icmph;
        struct igmphdr                  *igmph;
        struct iphdr                    *ipiph;
        struct ipv6hdr                  *ipv6h;
        unsigned char                   *raw;
        } h;

        union {
        struct iphdr                    *iph;
        struct ipv6hdr                  *ipv6h;
        struct arphdr                   *arph;
        unsigned char                   *raw;
        } nh;
        union {
        struct ethhdr                   *ethernet;
        unsigned char                   *raw;
        } mac;

        struct dst_entry                *dst;

        char                            cb[48];

        unsigned int                    len, data_len,
                                        mac_len,csum;
```

```
        unsigned char           local_df, cloned,
                                pkt_type, ip_summed;
        __u32                   priority;
        unsigned short          protocol,security;
        void                    (*destructor)(struct sk_buff *skb);

        unsigned char           *head,*data,
                                *tail, *end;

        ...
};
```

- *head* and *end* are pointers to the beginning and end of the occupied memory in the RAM.
- *data* and *tail* are pointers to the beginning and end of the buffer in the RAM. The pointers are different than *head* and *end* because they manage extra space to avoid copy instructions when additional/longer headers are added to the packet.
- *mac* points to the beginning of the MAC-header while *nh* and *h* are pointers to the header of the routing layer and transport layer respectively.
- *stamp* holds the arrival time of the packet.
- *dev* points to the network device that is associated with the packet.
- *sk* holds the socket to which the packet belongs.
- *dst* defines the destination or the next hop.
- *next* and *previous* are used to maintain a doubled linked list.

### A.2.2 Network device structure net_device

The header file in which the *net_device* structure is defined is **include/linux/netdevice.h**. Each network device is represented with this structure. A structure needs to be registered when its fields are filled with the values of a device. This is accomplished with the help of the function *register_netdevice()* in **drivers/net/net_init.c**. The *net_device* structure is quite large, therefore, the following extract focuses on the most important fields only.

```
struct net_device {
        char                name[IFNAMSIZ];
        unsigned long       mem_end; /* shared mem end */
        unsigned long       mem_start; /* shared mem start */
        unsigned long       base_addr; /* device I/O address */
        unsigned int        irq; /* device IRQ number */
        unsigned long       state;
        struct net_device   *next;
        /* The device initialisation function. Called only once. */
        int                 (*init)(struct net_device *dev);
        /* ——- Fields preinitialized in Space.c finish here ——- */
        struct net_device   *next_sched;
```

```
        int                     ifindex;
        struct net_device_stats* (*get_stats)(struct net_device *dev);
        struct ethtool_ops      *ethtool_ops;
        unsigned long           trans_start; /*Time (in jiffies) of last Tx*/
        unsigned long           last_rx; /* Time of last Rx */
        unsigned short          flags; /* interface flags (a la BSD) */
        unsigned                mtu; /* interface MTU value */
        unsigned short          type; /* interface hardware type */
        unsigned short          hard_header_len; /* hardware hdr length */
        void                    *priv; /* pointer to private data */
        /* Interface address info. */
        unsigned char           broadcast[MAX_ADDR_LEN]; /* hw bcast add */
        unsigned char           dev_addr[MAX_ADDR_LEN]; /* hw address */
        unsigned char           addr_len; /* hardware address length */
        void                    *atalk_ptr; /* AppleTalk link */
        void                    *ip_ptr; /* IPv4 specific data */
        void                    *dn_ptr; /* DECnet specific data */
        void                    *ip6_ptr; /* IPv6 specific data */
        void                    *ec_ptr; /* Econet specific data */
        void                    *ax25_ptr; /* AX.25 specific data */
        /* Called after device is detached from network. */
        void                    (*uninit)(struct net_device *dev);
        /* Pointers to interface service routines. */
        int                     (*open)(struct net_device *dev);
        int                     (*stop)(struct net_device *dev);
        int                     (*hard_start_xmit) (struct sk_buff *skb,
                                struct net_device *dev);
        int                     (*hard_header) (struct sk_buff *skb,
                                struct net_device *dev,
                                unsigned short type, void *daddr,
                                void *saddr, unsigned len);
        int                     (*rebuild_header)(struct sk_buff *skb);
        int                     (*set_mac_address)(struct net_device *dev,
                                void *addr);
        int                     (*do_ioctl)(struct net_device *dev,
                                struct ifreq *ifr, int cmd);
        int                     (*set_config)(struct net_device *dev,
                                struct ifmap *map);
        int                     (*change_mtu)(struct net_device *dev, int new_mtu);
        void                    (*tx_timeout) (struct net_device *dev);
        int                     (*hard_header_parse)(struct sk_buff *skb,
                                unsigned char *haddr);
        int                     (*neigh_setup)(struct net_device *dev,
                                struct neigh_parms *);
        int                     (*accept_fastpath)(struct net_device *,
                                struct dst_entry*);
        ...
};
```

- *mtu* (maximum transfer unit) defines the maximum length of a frame.
- *type* holds the type of the device, e.g. ARPHDR_IEEE802 for Ethernet 802.2. The constants

are defined in if_arp.h.

- *dev_addr* holds the MAC address.
- *hard_start_xmit* is used to send packets.
- *do_ioctl* is used to send *ioctl* commands.

### A.2.3 Socket structure

Linux uses the concept of kernel sockets. This means that for each socket opened by applications in the user space, there exists one instance of the structure *socket* and one instance of the structure *sock* in the kernel space. They are the interfaces to the upper and to the lower layers, respectively. The *socket* structure is defined in **include/linux/net.h**.

```
struct socket {
        socket_state        state;
        unsigned long       flags;
        struct proto_ops    *ops;
        struct file         *file;
        struct sock         *sk;
        short               type;
};
```

- *type* is the identification for the protocol type.
- *state* is the state of the socket. It can be free, unconnected, connecting, connected or disconnecting.
- *file* is a pointer to a file instance of a pseudo file. It is used in Linux (everything is a file) for communication.
- *socket* can be used for different protocols, therefore, it has with *proto_ops* a pointer to a structure that holds the protocol specific data.

The structure *sock* defined in **include/net/sock.h** contains the information that is important for the kernel. The definition of structure is more than 200 lines of code. The pointers are needed because these structures are stored in hash tables. The sending and receiving of packets is accomplished by queuing the packets into the *write_queue* and the *receive_queue* respectively. Some callback functions are associated with the structure to handle special events. The callback function *sk_data_ready()* is called when data can be delivered to the application layer.

```
struct sock {
        struct sock_common      __sk_common;
        struct sk_buff_head     sk_receive_queue;
        struct sk_buff_head     sk_write_queue;
        struct proto            *sk_prot;
        struct timer_list       sk_timer;
        /*Callbacks*/
        void                    (*sk_data_ready)(struct sock *sk, int bytes);
```

```
        ....
};
```

*sock_common* defined in **include/net/sock.h** is referenced by *sock*. It holds the protocol family and the state of the connection.

```
struct sock_common {
        unsigned short              skc_family;
        volatile unsigned char      skc_state;
        ...
};
```

## A.3 Datalink layer

The datalink layer receives the packets from the NIC (network interface card) and delivers them to the routing layer above it and vice versa. This section does not focus on the issues relating to the implementation of a driver, but only describes the interface to the driver. Figure A.2 shows the functions of the datalink layer and their interface to the upper layers.
The functions described in this section are defined or implemented in the following files:

- drivers/net/isa_skeleton.c
- include/linux/netdevice.h
- net/core/dev.c

### A.3.1 Receiving packets

Packets are received in an interrupt context. The interrupt service routine must not take too much time otherwise the performance of the entire system will degrade significantly. Therefore, only three simple tasks are performed during the interrupt processing:

- *net_interrupt()* implemented in **drivers/net/isa-skeleton.c** is the interrupt handler of the driver. If the interrupt is caused by an incoming packet then the packet is given to *net_rx()* for further processing. Otherwise this is an error.

- *net_rx()* implemented in **drivers/net/isa-skeleton.c** will create a new socket buffer. The packet is then directly copied into the memory of the kernel using the Direct Memory Access (DMA) technology. After that the header of the packet is examined to determine the identity of the protocol. The *dev* pointer of the *sk_buff* structure is set to the device over which the packet came in.

- *netif_rx()* will queue the packet in a queue of a processor. This function is, however, not driver specific. It is implemented in **/net/core/dev.c**.

Packets are managed in a *softnet_data* structure. Each processor has one *softnet_data* structure that will enhance the performance on multi-processor systems. It is defined in **include/linux/netdevice.h**.

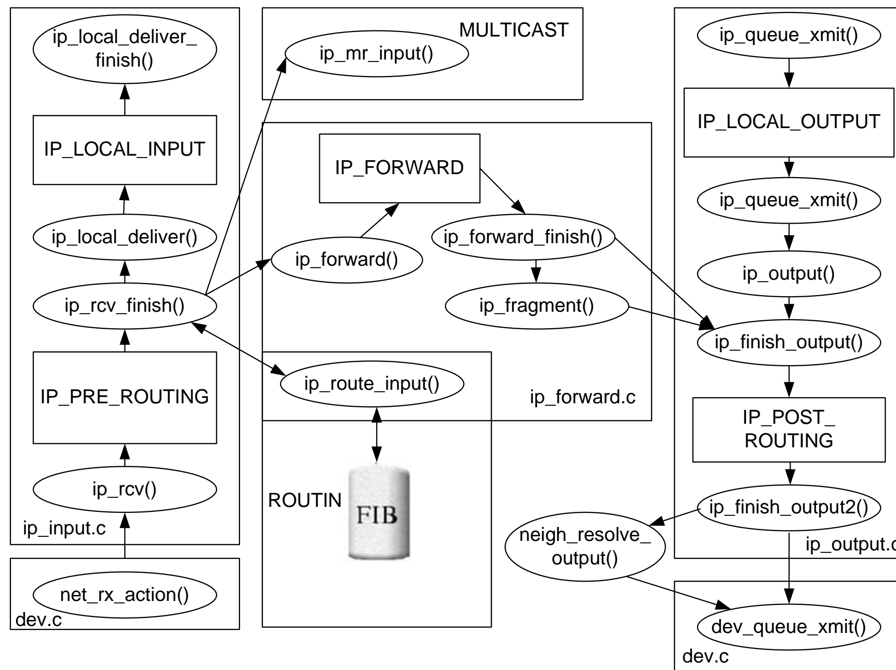



Figure A.2: Software architecture [222]

```
struct softnet_data {
        int                     cnb_level;
        int                     avg_blog;
        struct sk_buff_head     input_pkt_queue
}
```

- *input_pkt_queue* holds the *sk_buff_head* to manage the double linked list of incoming packets.

- *avg_blog* is the average length of the backlog. The backlog is defined as the packets that have arrived but are still not processed.

- *cng_level* is an integer that represents the congestion level.

When the packet is queued in the *softnet_data* structure, *netif_rx()* raises the *NET_RX_SOFTIRQ* software interrupt by calling the scheduler with a pointer to the *softnet_data* structure before leaving the interrupt context. Obviously, a defined software interrupt can not be called inside the interrupt context. *net_rx_action()* (also in **net/core/dev.c**) handles the task of this software interrupt. It checks the backlog in a circular fashion and gives the packet to the upper layer. To dequeue a packet, the function *__skb_dequeue()* is used. To figure out the type of the packet, *netif_receiver_skb()* is called. This function will determine the type field in the header and find the function to handle this type of packet (e.g. *ip_rcv()* for ipv4 packets). It sets the corresponding pointer of the *sk_buff* structure to that function and calls it.

**Sending packets**

To send a packet, the datalink layer has to receive a complete packet from the upper layers. All headers of the upper layers should have been inserted to identify it as a complete packet. Besides, the destination has to be in the *neighbour* structure because the datalink layer can not take routing decisions. To place a packet in the outgoing packet queue, *dev_queue_xmit()* defined in **net/core/dev.c** is called. The packet is then picked up after some time and sent through the function *hard_start_xmit()*. *hard_start_xmit()* can also be used directly with a socket buffer that carries the necessary information for the datalink layer and the outgoing interface in the form of a pointer to a *net_device* structure. *hard_start_xmit()* is implemented in **net/core/dev.c**.

## A.4 Network layer

The network layer is responsible for the processing of layer 3 protocols. Though the OSI model tries to split the functionality as strictly as possible, a couple of services are offered from the network layer to the datalink layer. The most important is routing because the datalink layer can not take decisions about choosing a next hop. Also the mapping of a MAC address to an IP address is handled in this layer. The functions described in this section are implemented in the following files:

- net/ipv4/ip_input.c
- net/ipv4/ip_forward.c
- net/ipv4/ip_output.c
- include/net/dst.h

### A.4.1 Receiving packets

The first function that is called in the network layer is the function *ip_rcv()* which is implemented in **net/ipv4/ip_input.c**. Its parameters are: the socket buffer received, the receiving device represented with a *net_device* structure and a *packet_type* structure with additional information about the datalink layer to find the right protocol. *ip_rcv()* first does some sanity checks, i.e. checking the length of the packet, checksums, the ipv4 or ipv6 version. The NetFilter hook **IP_PRE_ROUTING** has the opportunity to access the packet (see section A.8). In the next step, a decision has to be made, whether the packet is intended for the local host or has to be routed to a different host. *ip_route_input()* makes the routing decision. As a result of the routing decision, skb->dst->input points to the function that will be responsible for further processing. It points either to *ip_local_deliver()* or to *ip_forward()*.

**Receiving**

If the packet is intended for the local host, *ip_local_deliver()* has to find the correct transport layer port for the packet. *ip_local_deliver()* is implemented in **net/ipv4/ip_input.c** and it takes only one parameter that is the socket buffer structure. Usually, data payload is delivered to TCP or UDP. If the packet is not fragmented, the **NET_IP_LOCAL_IN** hook code is executed and then the processing continues in *ip_local_deliver_finish()*. Every protocol in the IP layer has an instance of *inet_protocol* with pointers to the functions that do either the further processing or handle errors. *ip_local_deliver_finish()* will strip off the IP header and will call the appropriate transport protocol handler function (e.g. *tcp_v4_rcv()* or *udp_rcv()*).

**Routing**

Two possibilities exist for a packet in the routing process:

- The destination is in the local network.
- The destination is a remote host outside the local network.

The latter case is more complex because next hop information is needed. This information can be found in the routing tables. The lookup already occurred in *ip_route_input()*, where the *dst* field of the socket buffer structure was set to the correct gateway. The work for *ip_forward()* is therefore straightforward:

1. if the TTL (Time to Live) field in the IP header is $\leq 1$ then drop the packet.
2. decrease the time to live (*ip_decrease_ttl()*) field.
3. call the function *ip_send_check()*.
4. give the packet to the forward NetFilter hook **NF_IP_FORWARD**.
5. call the function *ip_forward_finish()*.

*ip_forward_finish()* checks whether appropriate options are set in the IP header. If yes then *ip_forward_options()* takes care of them and calls *dst_output()*. *dst_output()* calls the function that is stored in the socket buffer as the handler function. The pointer to the handler function is in skb->dst->output (see details in section A.7).

### A.4.2 Sending

The function that is usually used to send data over IP is called *ip_queue_xmit()*. *ip_queue_xmit()* does the following steps:

1. make a routing decision if necessary.
2. call *ip_send_check()*.
3. give the packet to the outgoing hook **NF_IP_LOCAL_OUT**.
4. call *dst_output()*.

*ip_queue_xmit()* (implemented in **/net/ipv4/ip_output.c**) first chooses a route. Its parameters are a socket buffer containing the packet to be sent and a status bit that indicates whether the packet needs to be fragmented. To enhance performance, this lookup occurs only once for each socket (since they all have the same destination). *ip_send_check()* then calculates the checksum for the IP header. After the packet is returned from the NetFilter hook, *dst_output()* implemented in **/net/ipv4/ip_output.c** is called, which calls the function stored in skb->dst->output of the socket buffer. Usually, the pointer is set to *ip_output()*. If no fragmentation is necessary (packet size is $\leq$ MTU), *ip_finish_output()* (implemented in **/net/ipv4/ip_output.c**) is called with the socket buffer as parameter. *ip_finish_output()* will call the post routing hook **IP_POST_ROUTING**. After the hook, the processing continues in *ip_finish_output2()*. Here, the function checks whether enough memory is available for inserting the mac header. If not, more memory is allocated with *skb_realloc_headroom()*. The actual sending is handled by the function dst->neighbour->output. Usually, this pointer points to the function *dev_queue_xmit()*.

## A.5 UDP

UDP is a connectionless and unreliable protocol like IP.
The functions described in this section are defined or implemented in the following files:

- include/linux/udp.h
- include/linux/socket.h

- net/ipv4/af_inet.c
- net/ipv4/ip_output.c
- net/ipv4/udp.c
- net/core/datagram.c

In contrast to IP, the checksum is calculated on both header and payload. UDP is a simple protocol. The header consists only of source port, destination port, the checksum and information about the length of the packet (given in octets). The structure of UDP is defined in the header file **include/linux/udp.h**:

```
struct udphdr {
        __u16       source;
        __u16       dest;
        __u16       len;
        __u16       check;
};
```

### A.5.1 Data structures

For UDP, no complex data structures are used. This section describes the structures that are used to pass data over the UDP sockets between two hosts.

#### Passing data

The payload is given using the system call *sendmsg()* to the socket interface with the help of a *msghdr* structure defined in **include/linux/socket.h**. The data is checked and then copied into the kernel space. Then the structure is given to *udp_sendmsg()* without any further modifications.

```
struct msghdr {
        void                *msg_name;
        int                 msg_namelen;
        struct iovec        *msg_iov;
        __kernel_size_t     msg_iovlen;
        void                *msg_control;
        __kernel_size_t     msg_controllen;
        unsigned            msg_flags;
};
```

**struct msghdr** has the following parameters: *msg_name* is a pointer to a *sockaddr_in* structure that carries the IP address and a port number; *msg_namelen* holds the length of the *msg_name* structure. *msg_iov* references an array of *iovec* structures that hold pointers to the beginning and to the end of the data; *iov_len* is the length of the array. *msg_control* and *msg_controllen* specify buffers that are needed for storing control messages. The *msg_flags* are used to copy flags for packets from the kernel to the user space and from the user space to the kernel space. The following flags can be passed from the user space to the kernel space.

- *MSG_DONTROUTE* says that the destination is in the local subnet and must not be routed.
- *MSG_DONTWAIT* avoids a blocking system call if the data is not yet ready to be sent.
- *MSG_ERRROUTE* means that no packet should be fetched except an error message that might be waiting on the socket.

From kernel space to user space only *MSG_TRUNC* flag is passed. This flag indicates that enough memory does not exist for storing the data packet. Therefore, part of data might be lost.

**The UDP datagram**

The union element *h* of the *sk_buff* structure holds a pointer *struct udphdr *uh* that points to the UDP header within the packet. The structure holds the information about the UDP header.

**Integration of UDP in the network architecture**

UDP has two interfaces: one to the network layer and one to the application layer.

**The interface to IP** is defined through the *static struct inet_protocol udp_protocol* (defined in **net/ipv4/af_inet.c**). *udp_rcv()* delivers data to UDP, *udp_err* handles ICMP errors from the IP layer.

```
struct inet_protocol udp_protocol {
        .handler =          udp_rcf,
        .err_handler =      udp_err,
        .no_policy =        1,
        struct iovec        *iov;
        u32                 wcheck; };
```

To send data over IP, UDP uses the function *ip_build_xmit()* implemented in **net/ipv4/ip_output.c**. In contrast to *ip_queue_xmit()*, *ip_build_xmit()* does not contain the full IP relevant data but just a callback function that can access the necessary data.

**The interface to the application layer** is defined through a socket that accesses the features of the transport protocol through the *proto* structure defined for UDP in **net/ipv4/udp.c**:

```
struct proto udp_prot = {
        .name =             "UDP",
        .close =            udp_close,
        .connect =          udp_connect,
        .disconnect =       udp_disconnect,
        .iotcl =            udp_iotcl,
        .destroy =          udp_destroy,
        .setsockopt =       udp_setsockopt,
        .getsockopt =       udp_getsockopt,
        .sendmsg =          upd_sendmsg,
        .rcvmsg =           udp_rcvmsg,
        .sendpage =         upd_sendpage,
```

```
        .backlog_rcv =       udp_queue_rcv_skb,
        .unhash =            udp_v4_unhash,
        .get_port =          udp_v4_get_port,
};
```

- *udp_close()*: when a UDP socket is closed, *udp_close()* calls for all *PF_INET* sockets the function *inet_sock_release()* (implemented in **net/ipv4/af_inet.c**) to free the socket structure.
- *udp_connect()*: UDP is a connectionless protocol, therefore, this call is only made to define a socket. As a result, the packets can be routed quickly with the help of the routing cache entry for this socket. Besides, the state of the connection is set to established state in the packet header.
- *udp_disconnect()*: the state is set to *TCP_CLOSE*, destination address and port are deleted along with the corresponding routing cache entry.
- *udp_ioctl()*: The length of the waiting and sending queue can be enquired with the help of system call *ioctl()*.
- *udp_queue_rcv_skb*: see below in *udp_recvmsg()*.
- *udp_v4_hash()*: The received UDP packets must be queued to the correct socket. The hash makes the mapping faster. It is calculated using the formula: portnumber % *UDP_HTABLE_SIZE*.
- *udp_v4_unhash()* is used to remove the socket from the hash table.
- *udp_v4_get_port()*: this function is called from the *PF_INET* implemented in **net/ipv4/af_inet.c** when a port number is assigned to a socket. The port number may be NULL. In this case a free port is assigned.

### A.5.2 Functions

While the sending process occurs in one step, the receiving of a packet is split into two steps. After receiving the packet with *udp_rcv()*, it is queued in the receiving socket. From there, it is picked up by the corresponding user process with the *udp_recvmsg()* system call.

#### Sending

The function *udp_sendmsg()* (implemented in **net/ipv4/udp.c**) is always called if a UDP packet needs to be sent. The parameters are a *sock* structure with the state of the *PF_INET* socket and a pointer to the *msg* structure that holds the destination and the data.
If a socket is already established with *udp_connect()*, the packets do not have to carry the destination information. Instead, the information about the corresponding socket is only needed. Source address and source port are then extracted out of the *sock* structure. They are also available in a *ipcm_cookie* structure that will later provide this information to the IP layer.
Control messages of the *msg_control* element of the *msghdr* structure are handled by the *ip_cmsg_send()* function. The IP options can be passed with these control messages. If no additional information for IP is passed using the function then the default information from the *sock* structure is used.
If no route entry is in the routing cache or if a routing cache entry has become invalid then a new one is inserted with the help of *ip_route_output()* function.
The sending of a UDP packet itself is accomplished in *ip_build_xmit()* with a callback function as a parameter that is either *udp_getfrag()* or *udp_getfrag_nosum()* depending whether the checksum has to be calculated with the udp header or only for the payload.

**Receiving**

If a UDP packet is received by the IP protocol, it is passed as a pointer in the *sk_buff* structure to *udp_rcv()* (implemented in **net/ipv4/udp.c**). Here, the packet length and the checksum are checked. Then the packet is assigned to a socket and is inserted into its waiting queue. The insertion is done in *udp_rcv()* by calling the function *udp_v4_lookup()* which will look in the hash table for the corresponding socket. If a socket is found, *udp_queue_rcv_skb()* is called which in turn calls the function *sock_queue_rcv_skb()* that enqueues the packet in the correct queue. If no socket is found then an ICMP error message is generated.
*udp_recvmsg()* does the rest of the work. If *udp_recvmsg()* is called through the socket API function *recvmsg()* then it removes a *sk_buff* structure from the waiting queue and treats it like a UDP packet. If the waiting queue is empty then the calling process goes to sleep. Otherwise the payload is extracted from the packet and data is delivered to the user process.
The transport of *sk_buff* to the user process is is accomplished in *udp_recvmsg()* that calls following function defined in **net/core/datagram.c**:

- *skb_recv_datagram()* picks up the *sk_buff* structure by calling *skb_dequeue()*. If no packet is available in the queue then *wait_for_packet()* is called that in turn will call *schedule()* to relinquish control to the kernel.
- *skb_copy_datagram_iovec()* is called from *udp_recvmsg()* to copy the payload of the data packet in the *msg_iov* element of the *msghdr* structure.
- *skb_free_datagram()* will remove the *sk_buff* structure after the payload has been extracted.

## A.6 TCP

The TCP protocol has more features and is therefore more complex than UDP. TCP is a connection oriented reliable protocol. The payload is given to the socket in the right order.
The functions described in this section can be found in the following files:

- include/linux/tcp.h
- net/ipv4/tcp_ipv4.c

### A.6.1 TCP header

The TCP header holds the status information in the header. The structure of the header is defined in **include/linux/tcp.h** and looks like this:

```
struct tcphdr {
        __u16       source;
        __u16       dest;
        __u32       seq;
        __u32       ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16       res1:4, doff:4, fin:1, syn:1, rst:1,
                    psh:1, ack:1, urg:1, ece:1, cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16       doff:4, res1:4, cwr:1, ece:1, urg:1,
                    ack:1, psh:1, rst:1, syn:1, fin:1;
#else
#endif
```

```
        __u16        window;
        __u16        check;
        __u16        urg_ptr;
};
```

- *source* and *dest* are the port numbers each of two bytes long.
- *seq* is the sequence number. Every sent packet has a sequence number which is equal to the number of octets sent.
- *ack_seq* holds a number that is equal to the number of octets received.
- *doff* stands for *data offset* and is the length of the TCP header.
- *reserved* is not used because it is reversed by TCP.
- *urg*, *ack*, *psh*, *rst*, *syn* and *fin* are the control flags to handle the connection states.
- *window* tells the sender to transmit certain number of bytes before the next acknowledgment arrives.
- *check* is the checksum of the packet.
- *options* is a list of variable length.

### A.6.2 TCP states

A TCP connection is always in a certain state. The state can be *listen*, *connected* etc. Each state will make a transition to another state after taking a certain action.

CLOSED: A theoretical state where a TCP connection doesn't yet exist.
LISTEN: A state where TCP connection is waiting for a connection to be established with it.
SYN_RCVD: A SYN has been received and a SYN ACK has been sent, and the system is waiting for an ACK.
SYN_SENT: A SYN has been sent to start a connection but the SYN ACK yet has not been received.
ESTABLISHED: A connection is established.
CLOSE_WAIT: After a TCP connection is closed, the specification of TCP prohibits its reopening within few minutes. This mechanism prevents packets that were delayed in the Internet from reopening a socket.
LAST_ACK: A FIN has been received and one has been sent. Now, one has to wait for the ACK before moving to the closed state.
FIN_WAIT_1: A FIN has been sent, waiting for an ACK or FIN.
FIN_WAIT_2: A FIN has been sent and an ACK has been received. The communication could continue. This connection is half open.
CLOSING: A FIN has been sent and one FIN has been received but the FIN is still not acknowledged.

### A.6.3 Three way handshake

A connection has to be established before a TCP connection can be used to transmit data. In the TCP protocol two mechanisms are used for establishing a connection: *passive* or *active*. Before the connection gets established, the client socket is in state *closed* and the server socket is in state *listen*. The connection establishment happens by transmission of three packets:

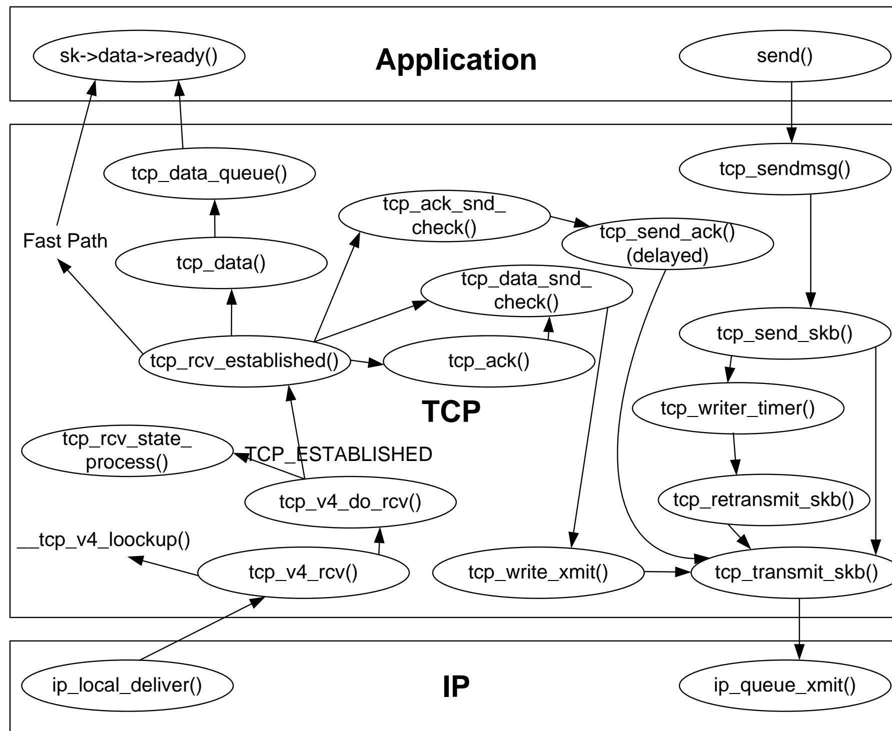



Figure A.3: TCP packet processing [222]

1. The client sends a *SYN* to the server. The socket status of the client switches from *CLOSED* to *SYN_SENT*.

2. The server receives the packet and returns a packet with the *SYN* and the *ACK* bits set. The status of the server changes from *CLOSED* to *SYN_REC*.

3. The client socket receives the packet with the *SYN* and the *ACK* bits set and changes it to status *ESTABLISHED*. Then, it sends an *ACK* packet back to the server.

4. The server receives the *ACK* packet and also changes to state *ESTABLISHED*.

### The passive connection establishment

The passive connection establishment is not initiated by the kernel but by a packet with a *SYN* flag. This packet is given to the function *tcp_v4_rcv()* (implemented in **net/ipv4/tcp_ipv4.c**). *tcp_v4_rcv()* finds the listening socket and hands over control to *tcp_v4_do_rcv()*. This function first calls *tcp_rcv_state_process()* to handle the states. If the socket is in the state *TCP_LISTEN*, the function *tcp_v4_conn_request()* is called. Besides many other things, the function sends an *ACK* packet at the end and switches to status *TCP_SYN_RCV*.

### The active connection establishment

The active connection establishment is initiated by a *socketcall* system call from the user space that in turn calls *tcp_v4_connect()*. Here, an IP route to the destination must be found. After the TCP header is generated, the connection status is set to *SYN_SENT*. Then a *SYN* packet is sent to the destination. The rest of the three-way-handshake is handled by *tcp_rcv_synsent_state_process()*.

## A.7 Routing

The routers forward IP packets from one subnet to others. Usually, the distance in hops decreases with every router that a packet visits on its way towards its destination. The router needs routing information to select a next hop for the data packet. This information is called forwarding information and is stored inside the *fib* (forward information base).
The functions described in this section are defined or implemented in the following files:

- net/ipv4/fib_rules.c
- net/ipv4/ip_fib.h
- net/ipv4/devinet.c
- net/ipv4/fib_hash.c
- net/ipv4/route.c
- include/net/ip_fib.h

A router has two important functions:

- gather *routing information*. This is accomplished by exchanging routing information with other routers to fill the *fib* with the necessary information.
- *routing* of IP packets. This action is done by consulting the *fib*. Afterward, the packet is sent according to the information in the *fib*.

The routing itself is the job of the IP layer and that is why it is implemented in the kernel. The *routing information* is collected by routing protocols. The interface between the routing and the routing protocol is the *fib*.

### A.7.1 Policy routing

Since kernel 2.4, routing decisions can depend not only on the destination address but also on a couple of other factors. An additional rule set selects an appropriate routing table to make the routing decision. The criteria can be the source address, the network device over which the packet came or the value of the Type of Service (TOS) field in the IP header.
Each rule has a selector and a type. The selector identifies the packets on which the rules are to be applied. The type defines what to do with this packet, i.e. use a certain routing table or to drop the packet. Each rule has an associated priority as well.
By default, the kernel has three unicast routing tables: local, main and default. The local table holds the routing information about the local devices. The tables main and default are initialized either by an administrator or a routing protocol. Using policy routing, other 252 tables can be created and linked to the ruleset.

### A.7.2 Implementation

This subsection describes the implementation of the routing rules, the routing table and the functions that manage the routing table.

#### Routing rules

In the policy routing, the routing rules decide in which order and in which table to look for the destination. The rules are traversed according to their priority. The implementation of the routing rules is in the file **net/ipv4/fib_rules.c**. If policy routing is not compiled into the kernel, this file will not be compiled. Instead, an inline function in **net/ipv4/ip_fib.h** will be used for the tables *local* and *main*.

**Data structures** The rule set is represented in the kernel as a list of *fib_rule* structures sorted by their priority. The head of the list is *fib_rules*. The *default_rule*, *main_rule* and *local_rule* rules are by default in the list. The list is protected with the read-write spinlock *fib_rules_lock*. The structure *fib_rule* contains two pointers to manage the linked list.

```
struct fib_rule {
        struct fib_rule *       r_next;
        u32                     r_preference;
        unsigned char           r_table;
        unsigned char           r_action;
        unsigned char           r_dst_len;
        unsigned char           r_src_len;
        u32                     r_src;
        u32                     r_srcmask;
        u32                     r_dst;
        u32                     r_dstmask;
        u32                     r_srcmap;
        u8                      r_flags;
        u8                      r_tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
        u32                     r_fwmark;
#endif
        int                     r_ifindex;
#ifdef CONFIG_NET_CLS_ROUTE
        __u32                   r_tclassid;
#endif
        char                    r_ifname[IFNAMSIZ];
        int                     r_dead;
};
```

- *u32 r_preference* is the priority.
- *unsigned char r_table* is a reference to the routing table that is eventually used.
- *unsigned char r_action* is the action to be taken (e.g. unicast, prohibit, nat (network address translation), ...).
- *u32 r_srcmap* and *__u32 r_tclassid* are used for nat.

The rest of the structure elements are used for the selection part of the rule. Here, the source address, the destination address, the network masks and TOS fields and the name of network interface are stored.

**RT-Netlink** is the interface to manage the routing rules. The table *inet_rtnetlink_table[]* contains pointers to the functions that are used for adding and deleting rules. The table is implemented in **net/ipv4/devinet.c**. For the messages *RTM_NEWRULE*, *RTM_GETRULE* and *RTM_DELRULE*, the functions *inet_rtm_newrule()*, *inet_dump_rules()* and *inet_rtm_delrule()* are called.

**Interface to routing tables** is primarily accessed through *fib_lookup* implemented in **fib_rules.c** because the routing rules are the gateways to the routing tables. *fib_lookup ()* will return the route to a key (*rt_key* structure) consisting of source address, destination address, incoming and outgoing interface and the TOS value. If *fib_lookup ()* returns without success, the function *fib_select_default()* is called that chooses a default route.

### Routing tables

Routing tables are represented within the kernel as complex data structures. The routing table itself is a **fib_table** structure. Up to 255 routing tables are available if the policy based routing is employed. For each length of the prefix, the routing table has a pointer to a **fn_zone** structure. In the **fn_zone** structure, the routing rules of type **fib_node** structure are stored. The actual data of a routing rule is stored in a **fib_info** structure which is referenced by **fn_node**. The structure is shown in Figure A.4.

**struct fib_table** defined in **include/net/ip_fib.h** has besides its id (*tb_id*) and the unused field *tb_stamp* the following fields:

```
struct fib_table{
        unsigned char  tb_id;
        unsigned       tb_stamp;
        int            (*tb_lookup)(struct fib_table *tb, const struct flowi *flp,
                              struct fib_result *res);
        int            (*tb_insert)(struct fib_table *table, struct rtmsg *r,
                              struct kern_rta *rta, struct nlmsghdr *n,
                              struct netlink_skb_parms *req);
        int            (*tb_delete)(struct fib_table *table, struct rtmsg *r,
                              struct kern_rta *rta, struct nlmsghdr *n,
                              struct netlink_skb_parms *req);
        int            (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
                              struct netlink_callback *cb);
        int            (*tb_flush)(struct fib_table *table);
        void           (*tb_select_default)(struct fib_table *table,
                              const struct flowi *flp, struct fib_result *res);

        unsigned char tb_data[0];
};
```

- *tb_insert* and *tb_delete* are used to insert and remove entries.
- *tb_dump* dumps the entries over the RT-Netlink.
- *tb_lookup* searches the table for a certain key. It is mainly used by *fib_lookup*.
- *tb_flush* deletes all entries marked for deletion.
- *tb_select_default* chooses one of the multiple default routes.

**struct fn_zone** holds all entries with the same length of the prefix in a hash table (**net/ipv4/fib_hash.c**). The hash table is an array of *fib_node* structures. This zone is only created if entries with a specific prefix length exist. Since the routing algorithm tries to find the route with the longest prefix first, therefore, it traverses this array of *fn_zones*.

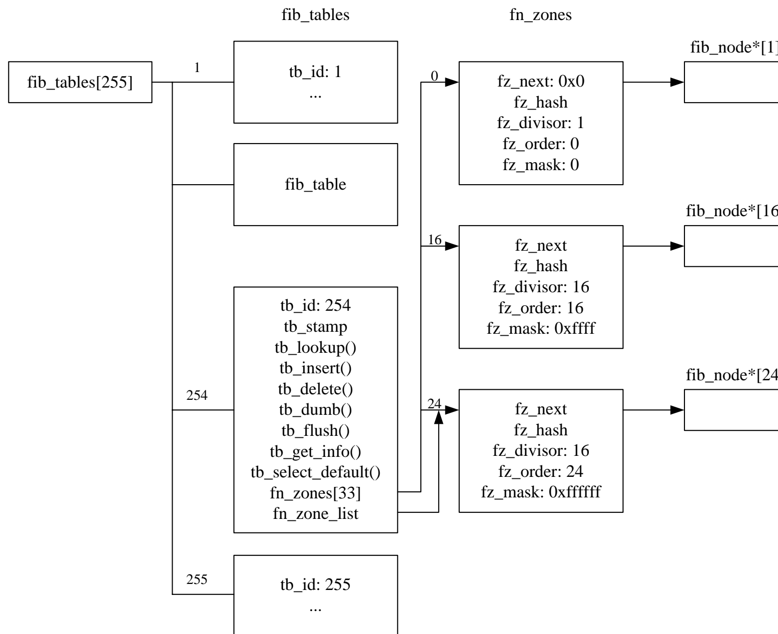



Figure A.4: *fib_table structure [222]*

**struct fib_node** is an entry in a routing table (**net/ipv4/fib_hash.c**). In the *fn_key* element a prefix is stored which is the same for all *fn_zone* structures. *fn_tos* holds the TOS value which can influence a routing decision. Type and scope are coded in *fn_type* and *fn_scope*. A routing entry could be found via an *fn_info* pointer in a *fib_info* structure (see figure A.5).

**struct fib_info** is the result of a FIB query (defined in **include/net/ip_fib.h**). The main information stored here is the outgoing device and the next hop on the way to the destination. This information is found in a *fib_nh* structure of the *fib_info*. Multiple *fib_nh* structures are stored in an array to enable multi-path routing to a destination. The number of possible paths is stored in *fib_nhs* of the *fib_info* structure. The *fib_nh* structure is defined in **ip_fib.h** and contains the outgoing device (*nh_oif*), a pointer to the *net_device* structure (*nh_dev*), and the IP address of the next router (*nh_gw*). The pointers *fib_next* and *fib_previous* are used to maintain a double linked list. They are used to insert new entries and are tied to the head *fib_info_list*.

### Functions

As previously mentioned, *ip_rcv_finish()* calls the function *ip_route_input()* to find the next hop on the path towards the destination of a packet.

**ip_route_input()** is called for every packet coming in through a network device (**net/ipv4/route.c**). Parameters are a pointer to a socket buffer, the source and destination addresses of the packet, the TOS value and a pointer to the *net_device* structure, through which the packet entered the current node. If the packet is unicast and no route was found in the cache then the function *ip_route_input_slow()* is called.

**ip_route_input_slow()** (**net/ipv4/route.c**) fills a *rt_key* structure with its parameters before starting a query with *fib_lookup()*. If this query is not successful, *ip_rcv_finish()* will discard the packet later. If the query is successful then the system can distinguish whether the packet is intended for the local host or for another host. If more than one route to the destination exist then *fib_select_multipath()* will choose one randomly according to the weights given. Then a new

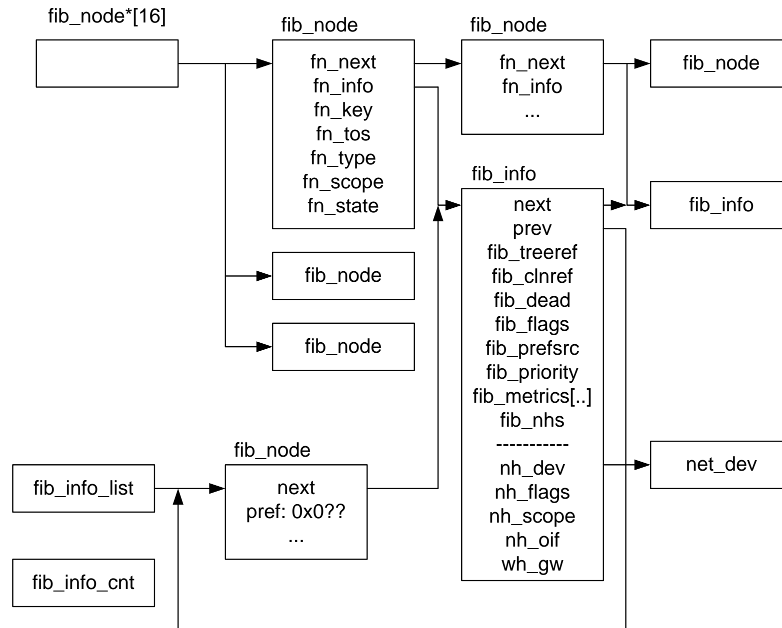



Figure A.5: *fib_node structure [222]*

cache entry is inserted with a pointer to the *output()* function. Additionally, *rt_gateway* is set using the function *rt_set_next_hop()*. The *dst* pointer of the *sk_buff* structure points to *dst_entry* structure, as a result of calling *ip_route_input()/ip_route_input_slow()*. The structure *dst_entry* looks like:

```
struct dst_entry {
        struct net_device     *dev;
        int                   (*input)(struct sk_buff*);
        int                   (*output)(struct sk_buff*);
        struct neighbour      *neighbour;
        ...
};
```

- *input* and*output* are the functions that process incoming or outgoing packets.
- *dev* specifies the network device that is used to process the outgoing packet. A packet for the local host has its *input* pointer pointing to *ip_local_deliver()* and *output* points to *ip_rt_bug*. For a packet that has to be routed, *input* points to *ip_forward()* and *output* points to *ip_output()*.

The neighbour field is used to address packets that are destined for a host within the local subnet. They can be reached within the datalink layer.

```
struct neighbour {
```

```
        struct net_device     *dev;
        unsigned char         ha[MAX_ADDR_LEN]
        int                   (*output)(struct sk_buff *skb)
        ...
};
```

*dev* points to the outgoing network device and *ha* is the hardware address of the destination device. *output* is a pointer to the function that is used to send the packet. Neighbour instances are created by the Address Resolution Protocol (ARP) layer.

## A.8 NetFilter

NetFilter is a framework inside the Linux kernel that gives the opportunity to mangle or filter packets. The idea of NetFilter is to give the writer of a module the chance to influence the processing of a packet in a very flexible way without the need to make changes in the network stack of the kernel. NetFilter gives a chance to do detailed network analysis and create dynamic filters of different kinds.
The functions described in this section are defined or implemented in the following files:

- include/linux/netfilter.h
- include/linux/netfilter_ipv4.h
- net/ipv4/in_forward.c
- net/core/netfilter.c

As mentioned in the previous sections that there are NetFilter hooks inside the kernel. NetFilter hooks work in the following way:

- **hooks** are implemented in the stack and can execute the NetFilter code.
- **NetFilter modules** are called from the hooks of the network stack. But they are independent from the code. Some standard modules exist that offer common functionality but user specific functions can also be implemented.

### A.8.1 Calling hook functions

Within the function of the network layer, hooks are placed to process NetFilter code. These hooks split a function into two (e.g. *ip_forward()* and *ip_forward_finish()*). This architecture is chosen because the kernel can also be compiled without NetFilter support. To avoid pre-compiler directives in the kernel code the NetFilter hooks are not called within a function. A NetFilter hook is called from the macro **NF_HOOK** defined in <**netfilter.h**> if compiled into the kernel. The arguments of the macros are:

```
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn)
(list_empty(&nf_hooks[(pf)][(hook)]
? (okfn)(skb)
:nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn))
#endif
```

- **pf** is the protocol family for which the hook calls the NetFilter code.
- **hook** is a number that references the hook. It is defined in <**netfilter_ipv4.h**> in IPv4.
- **skb** is the current socket buffer.
- **indev** and *outdev* are pointers to *net_device* instances through which the packet came and through which it will leave. The *outdev* pointer may be NULL before the routing decision is made.
- **okfn** is a pointer to a function that is called when the hook is finished.

*nf_hooks* is a global variable holding all NetFilter hooks of a certain protocol family that are sorted by the hook number. When processing **NF_HOOK**, it is checked whether a hook exists for the current protocol on the current position. If a hook exists, *nf_hook_slow()* is called to process the code. Otherwise, the processing continues with the function defined in *okfn*. NetFilter hooks are called in the functions *ip_rcv()*, *ip_forward()* and *ip_finish_output()* for a packet that has to be routed. In the function *ip_forward()* the **IP_FORWARD** hook is called as shown in the following. It is the last line of the function (see **net/ipv4/in-forward.c**):

```
{
        ...
        return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev,
                rt->u.dst.dev, ip_forward_finish);
}
```

In this case, the *ok_fn* pointer is set to *ip_forward_finish()*.

### A.8.2 Searching the hook table

*nf_hook_slow()* is called if there is at least one hook function registered. For each protocol family, there exists a double linked list containing instances of *nf_hook_ops*. This list is implemented in **net/core/netfilter.c**:

```
struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS];
```

NPROTO is a number that defines how many different protocols may be used (currently 32). For each protocol NF_MAX_HOOKS can be defined (currently 8). *nf_hook_ops* are defined in **/include/linux/netfilter.h** and look like this:

```
struct nf_hook_ops {
        struct list_head        list;
        nf_hookfn               *hook;
        struct module           * owner;
        int                     pf;
```

```
        int             hooknum;
        int             priority;
        ...
};
```

While *list* is a link to the list head, the other elements represent the following information:

- *hook* is a pointer to the hook function. This function needs the same parameters as the **NF_HOOK** macro.
- *pf* and *hooknum* hold the protocol family and the hook number. This can also be obtained from the position of the structure.
- *priority* is used to define a priority for the hook. Here all signed integers can be used.

Using this list, the matching hook can be called. The further processing is done by *nf_iterate* which traverses the list and calls the hook functions.

### A.8.3 Actions of hook functions

Each hook function will return one of the following values:

- **NF_ACCEPT** accepts a packet. This means that the current routine won't change the data in any way.
- **NF_STOLEN** tells the hook function that a packet is stolen and that it will be processed somewhere else. This means that the packet will not go through the other hooks and it will not be handed to another layer.
- **NF_DROP** tells the kernel to drop the packet. Like in **NF_STOLEN**, the packet is not processed in any way. Additionally, the socket buffer structure will be deleted and its memory is deallocated.
- **NF_QUEUE** will queue the packet in a waiting queue that will deliver the packet to the user space without processing any further hooks.
- **NF_REPEAT** will repeat the same hook again, therefore, one should carefully avoid loops.

The NetFilter is complemented by *iptables* that makes its use easier. Moreover, it could be used from the user space.

## A.9 Nature inspired routing protocols in the Linux kernel

This section focuses on the main files and functions in the Linux kernel that are helpful to implement Nature inspired routing protocols. All Nature inspired routing protocols have three common properties:

- Exchange routing information (Agent propagation)
- Queue management
- Quality evaluation

### A.9.1 Agent propagation

Agent propagation in general means sending and receiving agents that carry the routing information needed to make routing decisions. This task can be easily accomplished with UDP packets.

**Sending agents**

Agents can be launched by following the steps as shown in the box below. First, a UDP socket is created with *sock_create()* (implemented in **/net/socket.c**). Then the receiver is defined with a *sockaddr_in* structure (implemented in **include/linux/in.h**). *struct msghdr* holds the relevant parameters (see section UDP). Its field *msg_flags* is set to '0' before sending (see [117]).

```
struct socket *clientsocket = NULL;
char buf[64];
struct msghdr msg;
struct iovec iov;
struct sockaddr_in to;

/* SOCK_DGRAM, IPPROTO_UDP make a UDP socket */
sock_create(PF_INET, SOCK_DGRAM, IPPROTO_UDP, &clientsocket);
to.sin_family= AF_INET;
to.sin_saddrṡ_addr = in_action("w.x.y.z");
to.sin_port = htons((unsigned short) BeeSERVERPORT);
msg.msg_name = &to;
msg.msg_namelen = sizeof(to);
memcpy(buf,"message",8));
iov.iov_base = buf;
iov.iov_len = 8; /* 8 Bytes */
msg.msg_control = NULL;
msg.msg_controllen = 0;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
sock_sendmsg(clientsocket, &msg, 8); /*sock_sendmsg() in /net/socket.c*/
```

**UDP server**

The server that receives the agents is relatively complex. It does not connect to another service but has to bind itself to a port using the *bind()* function. A thread has to be created that listens on the port. The server needs a *struct iovec* for the address, the size of the data buffer, a *struct sockaddr_in* for the client data (e.g. IP address and port) and a *struct msghdr*. The contents of the *struct iovec* is modified by the kernel so they have to be initialized each time before calling *sock_recvmsg()*. This is done in a *while loop*.

```
sock_create(PF_INET, SOCK_DGRAM, IPPROTO_UDP, &udpsocket);
udpsocket->ops->bind(udpsocket, (struct sockaddr *) &server, sizeof(server));
thread_pid = kernel_thread(thread, NULL, CLONE_KERNEL);
```

```
static int com_thread(void *data) {
    unsigned char buffer[100];
    while (!signal_pendign(current)){
        iov.iov_base = buffer;
        iov.iov_len = sizeof(buffer);
        sock_recvmsg(udpsocket, &msg, sizeof(buffer),0);
    }
    complete(&threadcomplete);
```

```
};
```

### A.9.2 Queue management

The Linux TCP/IP stack offers the possibility to schedule outgoing packets. This is required because different sessions have different requirements. For example, a FTP transfer needs more bandwidth and will not care if a single packet gets delayed. In a *Voice Over IP* (VoIP) session, bandwidth consumption is low but latency plays an important role. The different strategies to schedule packets in Linux are called queuing disciplines. Each Network Interface Card (NIC) has a queuing discipline. A queuing discipline is an algorithm that is transparent to the system and it decides which packet to send next. Packets can be stored in one or more queues. The queuing algorithms can be *pfifo_fast*, the *Token Bucket Filter* or *Stochastical Fairness Queuing*. They are all implemented in **net/sched/sch_*.c**. By default, Linux uses the *pfifo_fast* queuing discipline. *pfifo_fast* consists of three queues ordered by their priority. The packets in the second queue are only sent when the first queue is empty. Similarly, the packets in the third queue are only sent when both queue one and queue two are empty. The queuing disciplines *bfifo* and *pfifo* also employ a similar policy, however, they store statistic information about the number of bytes sent and the number of bytes that are currently in the queue. *bfifo* stores this information in bytes while *pfifo* stores it as the number of packets. To change the queuing discipline, the command line tool *tc* can be employed.

### A.9.3 Quality evaluation

For routing decisions, the different kinds of delays play an important role. Delays can be queuing delay, processing delay, transmission delay and propagation delay. This subsection introduces few methods to calculate them.

**Queuing delay** is calculated with the help of following formula:

$$\frac{NumberOfBitsInQueue}{Bandwidth}$$

The bandwidth of a link can be accessed in the *type* field of the *net_device* structure, e.g. ARPHRD_ETHER stands for 10 Mbit/s. The meaning of the constants found here are defined in **include/linux/if_arp.h**. Since kernel 2.6.10, a *rate estimator* is available that holds the information about how many bits per second are processed. It is defined in header file **include/linux/gen_stats.h**. The structure containing the information can be accessed over the *qdisc* pointer of the used network device. The number of bits in the queue can be accessed using the statistics of the *bfifo*/*pfifo* queues as described in the previous sub section A.9.2.

**Processing delay** consists of steps such as looking up a route and changing a header. NetFilter hooks could be used to calculate the processing delay. The hooks *PREROUTING* and *POSTROUTING* are the first and the last steps in the processing of a packet. A packet that has to be routed can be picked up by the *PREROUTING* hook. Then the hook must timestamp this packet. The *POSTROUTING* hook than simply takes a difference of the current time and the time stamp of packet to calculate the processing delay.

**Transmission delay** is the amount of time it takes to put a packet on the transmission link. Transmission delay is determined by the bandwidth of the link and the size of the packet. Both can be easily determined using the previously mentioned techniques. The formula is:

$$\frac{LengthofPacketInBits}{BandwidthInBitsPerSecond}$$

**Propagation delay** is the amount of time it takes a packet to travel the distance on a transmission link between two nodes. This is essentially controlled by the speed of the signal inside the transmission medium and is independent of the networking technology used. As a rule of thumb, it takes about 20ms to send information over a distance of 1000 kilometers. To calculate the propagation delay, the time when the packet leaves a node and its arrival time at its neighbour need to be determined. When a packet is transmitted an interrupt is raised by the NIC that reports the successful transmission. The interrupt also reports a time stamp. When a packet arrives, the function *netif_rx()* creates a socket structure for the incoming socket and puts a time stamp in *skb->time*. The propagation delay can be calculated with the help of following formula after the packet has returned from the target:

$$PropagationDelay \equiv \frac{\Delta t - (2 * TransmissionDelay) - (ProcessingDelay)}{2}$$

Here, $\Delta t$ is the round trip time.

# References


[1] Packet filtering nat and packet mangling for linux. http://www.netfilter.org.

[2] ITU G.711: Pulse code modulation (pcm) of voice frequencies, November 1988.

[3] In H.-P. Schwefel, I. Wegener, and K. Weinert, editors, *Advances in Computational Intelligence - Theory and Practice*, Natural Computing Series. Springer-Verlag, 2003.

[4] *VoIP Standards and Protocols*. Faulkner Information Services, 2003.

[5] R. Ahuja, S. Keshav, and H. Saran. Design, implementation, and performance measurement of a native-mode atm transport layer (extended version). *IEEE/ACM Transaction on Networks*, 4(4):502–515, 1996.

[6] S. Appleby and S. Steward. Mobile software agents for control in telecommunications networks. *BT Technology Journal*, 12(2):104–113, apr 1994.

[7] A. F. Atlasis, M. P. Saltouros, and A. V. Vasilakos. On the use of a stochastic estimator learning algorithm to the atm routing problem: a methodology. *Computer Communications*, 21:538–546, 1998.

[8] S. Avallone, M. D'Arienzo, M. Esposito, A. Pescapé, S. P. Romano, and G. Ventre. Mtools. *IEEE Networks Magazine - Software Tools for Networking*, 16(5):3, october 2002.

[9] S. Avallone, D. Emma, A. Pescapé, and G. Ventre. A distributed multiplatform architecture for traffic generation. In *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, San Jose, California (USA), July 2004.

[10] S. Avallone, A. Pescapé, and G. Ventre. Distributed internet traffic generator (d-itg): analysis and experimentation over heterogeneous networks. In *ICNP 2003 poster Proceedings, International Conference on Network Protocols 2003*, Atlanta, Georgia (USA), November 2003.

[11] S. Avallone, A. Pescapé, and G. Ventre. Analysis and experimentation of internet traffic generator. In *New2an'04, Next Generation Teletraffic and Wired/Wireless Advanced Networking*, pages 70–75, 2004.

[12] D. Awduche and A. Chiu(et.al). RFC 3272: Overview and principles of internet traffic engineering, May 2002.

[13] P. Baldi, P. Frasconi, and P. Smyth. *Modeling the Internet and the Web: Probabilistic Methods and Algorithms*. Wiley, 2003.

[14] B. Baran and R. Sosa. A new approach for antnet routing. In *Ninth International Conference on Computer Communications and Networks*, pages 303–308, Las Vegas, NV, USA, 2000.

[15] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.

[16] H. G. Beyer and H. P. Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[17] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence From Natural to Artificial Systems*. Oxford University Press, 1999.

[18] E. Bonabeau, M. Dorigo, and G. Theraulaz. Inspiration for optimization from social insect behaviour. *Nature*, 406:39–42, 2000.

[19] E. Bonabeau, F. Hénaux, S. Guérin, D. Snyers, P. Kuntz, and G. Theraulaz. Routing in telecommunications networks with ant-like agents. In *Intelligent Agents for Telecommunication Applications, Second International Workshop, IATA '98, Paris, France, July 1998, Proceedings*, volume 1437 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1998.

[20] E. Bonabeau, G. Theraulaz, J. Deneubourg, S. Aron, and S. Camazine. Self-organization in social insects. *Trends in Ecology and Evolution*, 12(5):188–193, May 1997.

[21] Grady Booch. *Object-oriented Analysis & Design with Applicatiions*. Addison Wesley Longman Inc, 1994.

[22] A. Botta, D. Emma, S. Guadagno, and A. Pescapé. Performance evaluation of heterogeneous network scenarios. Technical report, Dipartimento di Informatica e Sistemistica, Universita' di Napoli Federico II, 2004.

[23] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in Neural Information Processing Systems*, 6:671–678, 1993.

[24] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the internet architecture: an overview, June 1994.

[25] F. M. T. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Desire: Modelling multi-agent systems in a compositional formal framework. *Int. J. Cooperative Inf. Syst.*, 6(1):67–94, 1997.

[26] L. T. Bui, Jürgen Branke, and A. Hussein. Multiobjective optimization for dynamic environments. In *Congress on Evolutionary Computation*. IEEE, 2006.

[27] S. Camazine, J. Deneubourg, N. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau. *Self-Organization in Biological Systems*. Princeton University Press, Princeton, NJ, second edition, 2003.

[28] K. Camp. *IP Telephony Demystified*. McGraw-Hill Companies Inc, 2003.

[29] D. Cantor and M. Gerla. Optimal routing in a packet-switched network. *IEEE Transactions on Computers*, 23:1062–1068, 1974.

[30] L. Carrillo, J. L. Marzo, L. Fàbrega, P. Vilà, and C. Guadall. Ant colony behaviour as routing mechanism to provide quality of service. In *Ant Colony Optimization and Swarm Intelligence, 4th International Workshop, ANTS 2004, Brussels, Belgium, September 5 - 8, 2004, Proceedings*, volume 3172 of *Lecture Notes in Computer Science*, pages 418–419. Springer, 2004.

[31] L. Carrillo, J. L. Marzoa, D. Harle, and P. Vila. A review of scalability and its application in the evaluation of the scalability measure of antnet routing. In Palau Salvador, editor, *IASTED Communication Systems and Networks CSN 2003*, pages 317–323, Benalmdena, Spain, 2003.

[32] W. Chainbi, M. Jmaiel, and A. B. Hamadou. Conception, behavioral semantics and formal specification of multi-agent systems. In *Multi-Agent Systems: Theories, Languages, and Applications, 4th Australian Workshop on Distributed Artificial Intelligence, Brisbane, Queensland, Australia, July 13, 1998, Selected Papers*, volume 1544 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 1998.

[33] D. D. Champeaux, D. Lea, and P. Faure. *Object-Oriented System Development*. Addison Wesley, 1993.

[34] J. Chen, P. Druschel, and D. Subramanian. An efficient multipath forwarding method. In *INFOCOM 98*, pages 1418–1425, 1998.

[35] J. Chen, P. Druschel, and D. Subramanian. A new approach to routing with dynamic metrics. In *INFOCOM 99*, pages 661–670, 1999.

[36] J. Chen, P. Druschel, and D. Subramanian. A simple, practical distributed multipath routing algorithm. TR98-320, Department of Computer Science, Rice University, July 1998.

[37] P. Choi and D. Yeung. Predictive q-routing: A memory-based reinforcement learning approach to adaptive traffic control. *Advances in Neural Information Processing Systems*, 8:945–951, 1996.

[38] Cisco. Internetworking technology handbook, 2002.

[39] D. Clark. RFC 817: Modularity and efficiency in protocol implementation. Technical report, MIT, July 1982.

[40] D. Clark. The structuring of systems using upcalls. *Operating Systems Review*, 19(5):171–180, 1985.

[41] D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM'90*, pages 200–208, Philadelphia, Sep 1990. ACM.

[42] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *IEEE Computer*, 37(2):48–51, 2004.

[43] Intel Corporation. Using the rdtsc instruction for performance monitoring. Application notes, pentium ii processor, Intel Corporation, 1997.

[44] A. Costa. Analytic modelling of agent-based network routing algorithms. Phd. thesis, The University of Adelaide, Australia, 2002.

[45] L. H. M. K. Costa, S. Fdida, and O. C. M. B. Duarte. Developing scalable protocols for three-metric qos routing. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 39(6):713–727, 2002.

[46] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, UK, 2001.

[47] J. Deneubourg, S. Aron, S. Goss, and J. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[48] G. Di Caro. Ant colony optimization and its application to adaptive routing in telecommunication networks. Phd. thesis, Université Libre de Bruxelles, Belgium, 2004.

[49] G. Di Caro and M. Dorigo. Antnet: A mobile agents approach to adaptive routing. Technical Report IRIDIA/97-12, Université Libre de Bruxelles, Belgium, 1997.

[50] G. Di Caro and M. Dorigo. An adaptive multi-agent routing algorithm inspired by ants behavior. In *In Proceedings of 5th Annual Australasian Conference on Parallel Real Time Systems*, pages 261–272, 1998.

[51] G. Di Caro and M. Dorigo. Ant colonies for adaptive routing in packet-switched communications networks. In *Parallel Problem Solving from Nature - PPSN V, LNCS 1498*, pages 673–682, Sept 1998.

[52] G. Di Caro and M. Dorigo. AntNet: Distributed stigmergetic control for communication networks. *Journal of Artificial Intelligence Research*, 9:317–365, December 1998.

[53] G. Di Caro and M. Dorigo. Extending AntNet for Best Effort Quality-of-Service Routing. In *First International Workshop on Ant Colony Optimization*, Brussels, Belgium, October, 15-16 1998.

[54] G. Di Caro and M. Dorigo. Mobile agents for adaptive routing. In *31st Hawaii International Conference on System Science*, pages 74–83, Big Island of Hawaii, 1998. IEEE Computer Society Press.

[55] G. Di Caro and M. Dorigo. Two ant colony algorithms for best-effort routing in datagram networks. In *Proceedings of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'98)*, pages 541–546. IASTED/ACTA Press, 1998.

[56] G. Di Caro and T. Vasilakos. Ant-SELA: Ant-agent and stochastic automata learn adaptive routing tables for qos routing in atm networks. In *ANTS2000 - From Ant Colonies to Artificial Ants: Second International Workshop on Ant Colony Optimization*, Sept 8-9 2000.

[57] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[58] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[59] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dmars. In *Intelligent Agents IV, Agent Theories, Architectures, and Languages, 4th International Workshop, ATAL '97, Providence, Rhode Island, USA, July 24-26, 1997, Proceedings*, volume 1365 of *Lecture Notes in Computer Science*, pages 155–176. Springer, 1998.

[60] S. Doi and M. Yamamura. Bntnetl: Evaluation of its performance under congestion. *IEICE B*, pages 1702–1711, 2000.

[61] S. Doi and M. Yamamura. Bntnetl and its evaluation on a situation of congestion. *Electronics and Communications in Japan*, 85:31–41, 2002.

[62] M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant algorithms and stigmergy. *Future Generation Computer Systems*, 16(8):851–871, June 2000.

[63] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.

[64] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[65] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy, 1991.

[66] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics-Part B*, 26(1):29–41, 1996.

[67] M. Dorigo and T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications and advances. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 251–285. Kluwer Academic Publishers, Norwell, MA, 2003.

[68] S. N. Dorogovtsev and J. F. F. Mendes. *Evolution of Networks: From Biological Nets to the Internet and WWW.* Oxford University Press, 2004.

[69] P. Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM'94*, pages 2–13, 1994.

[70] J. Durkin. *Voice-Enabling the Data Network.* CISCO Press, 2003.

[71] A. Edwards and S. Muir. Experiences implementing a high performance tcp in user-space. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 196–205, New York, NY, USA, 1995. ACM Press.

[72] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 14–23, New York, NY, USA, 1994. ACM Press.

[73] L. M. Feeney. A taxonomy for routing protocols in mobile ad hoc networks. Technical Report ISRN:SICS-T-99/07-SE, Swedish Institute of Computer Science, Kista, Sweden, 1999.

[74] S. Fenet and S. Hassas. Ant.: a distributed network control framework based on mobile agents. In *International ICSC Congress on Intelligent Systems And Applications*, pages 831–837. ICSC Academic Press Editor, 2000.

[75] S. Fenet and S. Hassas. A.n.t: a distributed problem-solving framework based on mobile agents. In *Mobile Agents Applications'2000 (12th International Conference On Systems Research, Informatic & Cybernetic)*, pages 39–44, 2000.

[76] J. E. Flood(Ed.). *Telecommunication Networks.* Publishing & Inspec, 1997.

[77] R. L. Freeman. *Telecommunication System Engineering.* John Wiley & Sons, Inc, 2004.

[78] E. Gafni and D. Bertsekas. Distributed algorithms for generating loopfree routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 29:11–18, 1981.

[79] R. Gallagher. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communications*, 25:73–85, 1979.

[80] M. Gallego-Schmid. Modified antnet: software application in the evaluation and management of a telecommunication network. In Una-May O'Reilly, editor, *Graduate Student Workshop*, pages 353–354, Orlando, Florida, USA, 13 July 1999.

[81] A. Giessler, J. D. Haenle, A. König, and E. Pade. Free buffer allocation - an investigation by simulation. *Computer Networks*, 2:191–208, 1978.

[82] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison Wesley, Reading, MA, 1989.

[83] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE parallel and distributed technology: systems and applications*, 1(3):12–21, 1993.

[84] P. P. Grassé. La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[85] R. S. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2002. Accepted for publication. Draft available as Technical Report TR2000-365, Department of Computer Science, Dartmouth College.

[86] R. Guerin, S. Blake, and S. Herzog. Aggregating RSVP-based qos requests, November 1997.

[87] A. Gupta and V. Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19(3):234–244, 1993.

[88] A. Harsch. Design and development of a network infrastructure for swarm routing protocols inside linux. Master thesis, LSIII, The University of Dortmund, Germany, July 2005.

[89] A. L. G. Hayzelden and J. Bigham. Agent Technology in Communications Systems: An Overview. *Knowledge Engineering Review*, 1999.

[90] C. L. Hedrick. RFC 1058: Routing information protocol, June 1998.

[91] T. Hendtlass and M. Ali, editors. *Collective Intelligence and Priority Routing in Networks*. Springer Verlag, 2002.

[92] J. L. Hennessey and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1995.

[93] M. Heusse, D. Syners, S. Guerin, and P. Kuntz. Adaptive agent-driven routing and load balancing in communication networks. *Advances in Complex Systems*, 1(2-3):237–254, 1998.

[94] G. N. Higginbottom. *Performace evaluation of communication networks*. Artech house Inc, Norwood, MA, 1998.

[95] V. Hilaire, A. Koukam, P. Gruer, and J. Pi. Müller. Formal specification and prototyping of multi-agent systems. In *Engineering Societies in the Agent World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000, Revised Papers*, volume 1972 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2000.

[96] J. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[97] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[98] C. A. Iglesias, M. Garijo, J. Centeno-González, and Juan R. Velasco. Analysis and design of multiagent systems using mas-common kads. In *Intelligent Agents IV, Agent Theories, Architectures, and Languages, 4th International Workshop, ATAL '97, Providence, Rhode Island, USA, July 24-26, 1997, Proceedings*, volume 1365 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 1997.

[99] P. Jain. Validation of antnet as a superior single path, single constrained algorithm. Master thesis, Department of Computer Science and Engineering, University of Minnesota, USA, 2002.

[100] N. R. Jennings. Agent-oriented software engineering in multi-agent system engineering. In *MultiAgent System Engineering, 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '99, Valencia, Spain, June 30 - July 2, 1999, Proceedings*, volume 1647 of *Lecture Notes in Computer Science*, pages 1–7. Springer, 1999.

[101] P. Jogalekar and C.M. Woodside. A scalability metric for distributed computing applications in telecommunications. In *Proceedings of the 15th International Teletraffic Congres*, pages 101–110, June 1997.

[102] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 524, Washington, DC, USA, 1998. IEEE Computer Society.

[103] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel Distributed Systems*, 11(6):589–603, 2000.

[104] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, May 1996.

[105] I. Kassabalidis, M. A. El-Sharkawi, and R. J. Marks. Adaptive-sdr: Adaptive swarm-based distributed routing. In *Proceedings of the 2002 International Joint Conference on Neural Networks, 2002 IEEE World Congress on Computational Intelligence*, pages 2878–2883, 2002.

[106] I. Kassabalidis, M. A. El-Sharkawi, R. J. Marks, P. Arabshahi, and A. A. Gray. Swarm intelligence for routing in communication networks. In *Global Telecommunications Conference GLOBECOM*, pages 3613–3617. IEEE, 2001.

[107] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2003.

[108] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 259–268, New York, NY, USA, 1993. ACM Press.

[109] F. Kelly. Network routing. *Philosophical Transactions of the Royal Society*, 337:343–367, 1991.

[110] S. Keshav. *An engineering approach to computer networking: ATM networks, the Internet, and the Telephone network*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[111] L. Kleinrock. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, volume 2, pages 27.2.1–27.2.5, Toronto, Canada, june 1978. IEEE.

[112] L. Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. In IEEE, editor, *Proceedings of the International Conference on Communications*, ICC, pages 335–347, France, 1979.

[113] Hartmut König. *Protocol Engineering (in german)*. Teubner, 2003.

[114] D. Kotz and R. S. Gray. Mobile agents and the future of the internet. *Operating Systems Review*, 33(3):7–13, 1999.

[115] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[116] K. Kümmerle and H. Rudin. Packet and circuit switching: Cost/performance boundaries. *Computer Networks*, 2:3–17, 1978.

[117] E. Kunst and J. Quade. Kern-technik, linux-magazin, October 2004.

[118] G. M. Lee and J. S. Choi. A survey of multipath routing for traffic engineering. Term paper, Informations and Communications University, Korea, 2002.

[119] S. Liang, A. N. Zincir-Heywood, and M. I. Heywood. The effect of routing under local information using a social insect metaphor. In *Proceedings of IEEE Congress on Evolutionary Computing*, May 2002.

[120] S. Liang, A. N. Zincir-Heywood, and M. I. Heywood. Intelligent packets for dynamic network routing using distributed genetic algorithm. In *Proceedings of Genetic and Evolutionary Computation Conference*. GECCO, July 2002.

[121] R. Love. *Linux Kernel Development*. Novel Press, 2 edition, 2005.

[122] C. Madukife. Development of a formal framework to analyze the behavior of swarm routing protocols. Master thesis, LSIII, The University of Dortmund, Germany, August 2005.

[123] G. Malkin. RFC 2453: Rip version 2, November 1998.

[124] V. Maniezzo and A. Carbonaro. Ant colony optimization: an overview. In C. Ribeiro, editor, *Essays and Surveys in Metaheuristics*, pages 21–44. Kluwer, 2001.

[125] A. Medina, , I. Matta, and J. Byers. On the origin of power laws in internet topologies. *ACM Computer Communication Review*, 30(2):18–28, April 2000.

[126] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: Universal topology generation from a user's perspective. Technical Report BU-CS-TR-2001-003, Boston University, 1 2001.

[127] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: Universal topology generation from a user's perspective. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS '01*. Cincinnati, Ohio, August 2001.

[128] D. Merkle, M. Middendorf, and A. Scheidler. Dynamic decentralized packet clustering in networks. In *Applications of Evolutionary Computing*, pages 574–583. Springer Verlag, April 2005.

[129] T. Michalareas and L. Sacks. Link-state & ant-like algorithm behaviour for single-constrained routing. In *IEEE Workshop on High Performance Switching and Routing, HPSR 2001*, pages 302–305, May 2001.

[130] T. Michalareas and L. Sacks. Stigmergic techniques for solving multi-constraint routing for packet networks. In Pascal Lorenz, editor, *Networking - ICN 2001, First International Conference, Colmar, France, July 9-13, 2001 Proceedings, Part 1 LNCS 2093*, pages 687–697. Springer-Verlag, 2001.

[131] D. L. Mills. RFC 958: Network time protocol (NTP), September 1985.

[132] N. Minar, K. H. Kramer, and P. Maes. *Cooperating Mobile Agents for Dynamic Network Routing*, chapter 12, pages 287–304. Springer-Verlag, 1999.

[133] J. Mogul. Ip network performance. In D. C. Lynch and M. T. Rose, editors, *Internet System Handbook*. Addison Wesley, 1993.

[134] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 39–51, New York, NY, USA, 1987. ACM Press.

[135] R. Mortier. Internet traffic engineering. Technical Report UCAM-CL-TR-532, University of Cambridge, Computer Laboratory, April 2002.

[136] C. Moschovitis, H. Poole, T. Schuyler, and T. Senft. *History of the Internet: A Chronology, 1843 to the Present*. ABC-CLIO, 1999.

[137] J. T. Moy. *OSPF Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1998.

[138] J. T. Moy. *OSPF Complete Implementation*. Addison-Wesley, 2000.

[139] M. Munetomo. Designing genetic algorithms for adaptive routing algorithms in the internet. In *Proceedings of GECCO'99 Workshop on Evolutionary Telecommunications: Past, Present and Future*. Orlando, Florida, July 1999.

[140] M. Munetomo. *Network Routing with the Use of Evolutionary Methods, in Computational Intelligence in Telecommunication Networks*. CRC Press, 2000.

[141] M. Munetomo, Y. Takai, and Y. Sato. An adaptive network routing algorithm employing path genetic operators. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 643–649. Morgan Kaufmann Publishers, 1997.

[142] W. Nachtigall. *Bionik; Grundlagen und Beispiele für Ingenieure und Naturwissenschaftler (german)*. Springer-Verlag, 2002.

[143] A. Newell. Physical symbol systems. *Cognitive Science*, 4:135–183, 1980.

[144] P. Nii. The blackboard model of problem solving. *AI Mag*, 7(2):38–53, 1986.

[145] K. Oida and A. Kataoka. Lock-free AntNet and its evaluation adaptiveness. *Journal of IEICE B (in Japenese)*, J82-B(7):1309–1319, 1999.

[146] K. Oida and M. Sekido. An agent-based routing system for qos guarantees. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 833–838, 1999.

[147] K. Oida and M. Sekido. ARS: An efficient agent-based routing system for qos guarantees. *Computer Communications*, 23:1437–1447, 200.

[148] E. Osborne and A. Simha. *Traffic Engineering with MPLS*. Cisco Press, 2002.

[149] S. Ossowski and A. García-Serrano. Social structure in artificial agent societies: Implications for autonomous problem-solving agents. In *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, ATAL '98, Paris, France, July 4-7, 1998, Proceedings*, volume 1555 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 1995.

[150] G. I. Papadimitriou. A new approach to the design of reinforcement schemes for learning automata: Stochastic estimator learning algorithms. *IEEE Trans. Knowl. Data Eng.*, 6(4):649–654, 1994.

[151] R. Pastor-Satorras and A. Vespignani. *Evolution and Structure of the Internet: A Statistical Physics Approach*. Cambridge University Press, 2004.

[152] L. L. Peterson and B. S. Davie. *Computer Networks A Systems Approach*. Morgan Kaufmann Publishers, 2000.

[153] J. Postel. RFC 768: User datagram protocol, August 1980.

[154] J. Postel. RFC 793: Transmission control protocol, September 1981.

[155] J. Postel and J. K. Reynolds. RFC 959: File transfer protocol, October 1985.

[156] G. N. Purdy. *Linux Iptables Pocket Reference*. O'Reilly & Associates, 2004.

[157] M. Rio, M. Goutelle, T.Kelly, R. Hughes-Jones, J. Martin-Flatin, and Y. Li. A map of the networking code in linux kernel 2.4.20, datatag, 2004.

[158] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, January 1965.

[159] R. D. Rosner. Circuit and packet switching. *Computer Networks*, 1:7–26, 1976.

[160] T. Rossmann and C. Tropea. *Bionik; Aktuelle Forschungsergebnisse in Natur-, Ingenieur- und Geisteswissenschaft (german)*. Springer-Verlag, 2004.

[161] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2002.

[162] T. N. Saadawi and M. H. Ammar. *Fundamentals of Telecommunication Networks*. John Wiley & Sons, Inc, 1994.

[163] H. G. Sandalidis, C. X. Mavromoustakis, and P. Stavroulakis. Ant based probabilistic routing with pheromone and antipheromone mechanisms. *Communication Systems*, 17:55–62, 2004.

[164] H. G. Sandalidis, C. X. Mavromoustakis, and P. P. Stavroulakis. Performance measures of an ant based decentralised routing scheme for circuit switching communication networks. *Soft Comput.*, 5(4):313–317, 2001.

[165] C. Santivanez, B. McDonald, I. Stavrakakis, and R. Ramanathan. On the scalability of ad hoc routing protocols. In *Proceedings of IEEE INFOCOM 2002*. IEEE, June 2002.

[166] S. R. Sarukkai, P. Mehta, and R. J. Block. Automated scalability anaylsis of message-passing parallel programs. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):21–32, 1995.

[167] R. Schoonderwoerd and O. Holland. Minimal agents for communications network routing: The social insect paradigm. *Software Agents for Future Communication Systems*, (1), 1999.

[168] R. Schoonderwoerd, O. Holland, and J. Bruten. Ant-like agents for load balancing in telecommunications networks. In *Agents*, pages 209–216, 1997.

[169] R. Schoonderwoerd, O. E. Holland, J. L. Bruten, and L. J. M. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5(2):169–207, 1996.

[170] T. D. Seeley. *The Wisdom of the Hive*. Harvard University Press, London, 1995.

[171] T. D. Seeley and W. F. Towne. Collective decision making in honey bees: how colonies choose among nectar sources. *Behavior Ecology and Sociobiology*, 12:277–290, 1991.

[172] R. Serfozo. *Introduction to Stochastic Networks*. Springer-Verlag, 1999.

[173] A. Silberschatz and P. B. Galvin. *Operating System Concepts (4th Edition)*. Addison-Wesley, 1994.

[174] K. M. Sim and W. H. Sun. Ant colony optimization for routing and load-balancing: Survey and new directions. *IEEE Transactions on Systems, Man and Cybernetics-Part A*, 33(5):560–572, 2003.

[175] H. A. Simon. *Administrative Behavior: A Study of Decision-making Processes in Administrative Organization*. Free Press, New York, 1976.

[176] M. C. Sinclair. Evolutionary telecommunications: A summary. In *Proceedings of GECCO'99 Workshop on Evolutionary Telecommunications: Past, Present and Future*. Orlando, Florida, July 1999.

[177] Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[178] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical report, College of Computing, Georgia Institute of Technology, Sept 1994. Tecnical Report GIT-CC-94/38.

[179] C. U. Smith. Designing high-performance distributed applications using software performance engineering: A tutorial. In *22nd International Computer Measurement Group Conference, December 10-13, 1996, San Diego, CA, USA, Proceedings*, pages 498–507. Computer Measurement Group, 1996.

[180] C. U. Smith and L. G. Williams. Building responsive and scalable web applications. In *26th International Computer Measurement Group Conference, December 10-15, 2000, Orlando, FL, USA, Proceedings*, pages 127–138. Computer Measurement Group, 2000.

[181] C. U. Smith and L. G. Williams. Performance and scalability of distributed software architectures: An SPE approach. *Parallel and Distributed Computing Practices*, 3(4), 2000.

[182] P. G. Spirakis and C. D. Zaroliagis. *Distributed algorithm engineering*, pages 197–228. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[183] R. E. Steuer. *Multiple Criteria Optimization: Theory, Computation and Application*. Wiley, New York, 1986.

[184] W. R. Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison Wesley, 1994.

[185] W. R. Stevens. *TCP/IP Illustrated: TCP for Transactions, HTTP, NTTP, and the UNIX Domain Protocols*, volume 3. Addison Wesley, 1996.

[186] W. R. Stevens. *UNIX Network Programming: Networking APIs – Sockets and XTI*, volume 1. Addison-Wesley, 1 edition, 1997.

[187] W. R. Stevens. *UNIX Network Programming, Interprocess Communication*, volume 2. Addison Wesley, 2 edition, 1999.

[188] P. Stone and M. M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Auton. Robots*, 8(3):345–383, 2000.

[189] T. Stützle and H. H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[190] Z. Subing and L. Zemin. A qos routing algorithm based on ant algorithm. In *Proceedings of IEEE International Conference on Communications (ICC'01)*, pages 1587–1591, 2001.

[191] D. Subramanian, P. Druschel, and J. Chen. Ants and reinforcement learning: A case study in routing in dynamic networks. In *Proceedings of 15th Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 832–839. Morgan Kaufmann, San Francisco, CA, 1997.

[192] D. J. T. Sumpter. From bee to society: An agent-based investigation of honey bee colonies. Phd. thesis, The University of Manchester, UK, 2000.

[193] X. H. Sun and L. M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993.

[194] S. Tadrus and L. Bai. A qos network routing algorithm using multiple pheromone tables. In *Web Intelligence*, pages 132–138, 2003.

[195] A. S. Tanenbaum. *Modern Operating Systems*. Internals and Design Principles. Prentice-Hall, International, 1992.

[196] P. Tarasewich and P. R. McMullen. Swarm intelligence: Power in numbers. *Communications of ACM*, 45(8):62–67, 2002.

[197] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 64–73, New York, NY, USA, 1993. ACM Press.

[198] M. Thirunavukkarasu. Reinforcing reachable routes. Master thesis, Virginia Polytechnic Institute and State University, 2004.

[199] R. A. Tintin and D. I. Lee. Intelligent and mobile agents over legacy, present and future telecommunication networks. In *First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99)*, pages 109–126. World Scientific Publishing Ltd., 1999.

[200] R. van der Put. Routing in packet switched networks using agents. Master thesis, KBS, Delft University of Technology, Netherlands, 1998.

[201] R. van der Put. Routing in the faxfactory using mobile agents. Technical report, KPN Research, 1998.

[202] S. Varadarajan, N. Ramakrishnan, and M. Thirunavukkarasu. Reinforcing reachable routes. *Computer Networks*, 43(3):389–416, 2003.

[203] A. Varga. OMNeT++: Discrete event simulation system: User manual. http://www.omnetpp.org.

[204] C. Villamizar. Ospf optimized multipath (ospf-omp). In *Proceedings of the fourty-fourth Internet Engineering Task Force*, INTERNET DRAFT, draft-ietf-ospf-omp-02, Minneapolis, MN, USA, February 1999.

[205] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of ACM Symposium on Operating systems principles*, pages 40–53, 1995.

[206] K. von Frisch. *The Dance Language and Orientation of Bees.* Harvard University Press, Cambridge, 1967.

[207] S. Vutukury. Multipath routing mechanisms for traffic engineering and quality of service in the internet. Phd. thesis, University of California, Santa Cruz, 2001.

[208] S. Vutukury and J. Garcia-Luna-Aceves. An algorithm for multipath computation using distance-vectors with predecessor information. In *Proceedings of 8th International Conference of IEEE Computer Communications and Networks*, pages 534–539. IEEE Press, 1999.

[209] S. Vutukury and J. Garcia-Luna-Aceves. A distributed algorithm for multipath computation. In *Proceedings of GLOBECOM*, pages 1689–1693, 1999.

[210] S. Vutukury and J. J. Garcia-Luna-Aceves. A simple approximation to minimum-delay routing. *ACM SIGCOMM Computer Communication Review*, 29(4):227–238, 1999.

[211] S. Vutukury and J. J. Garcia-Luna-Aceves. MDVA: A distance-vector multipath routing protocol. In *INFOCOM*, pages 557–564, 2001.

[212] C. J. Watkins. Learning from delayed rewards. Phd. thesis, Psychology Department, University of Cambridge, UK, 1989.

[213] C. J. Watkins and P. Dayan. Q-learning. *Machine Learning*, 1:279–292, 1992.

[214] R. W. Watson and S. A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, 1987.

[215] H. F. Wedde and M. Farooq. Beehive: An efficient, scalable, adaptive, fault-tolerant and dynamic routing algorithm inspired from the wisdom of the hive. Technical report, Informatik III, Computer Science Department, University of Dortmund, 2005. TR-801.

[216] H. F. Wedde and M. Farooq. A performance evaluation framework for nature inspired routing algorithms. In *Applications of Evolutionary Computing, LNCS 3449*, pages 136–146. Springer Verlag, March 2005.

[217] H. F. Wedde and M. Farooq. The wisdom of the hive applied to mobile ad-hoc networks. In *Proceedings of the IEEE Swarm Intelligence Symposium*, pages 341–348, 2005.

[218] H. F. Wedde, M. Farooq, T. Pannenbaecker, B. Vogel, C. Mueller, J. Meth, and R. Jeruschkat. BeeAdHoc: an energy efficient routing algorithm for mobile ad-hoc networks inspired by bee behavior. In *Proceedings of ACM GECCO*, pages 153–160, 2005.

[219] H. F. Wedde, M. Farooq, T. Pannenbaecker, B. Vogel, C. Mueller, J. Meth, R. Jeruschkat, M. Duhm, L. Bensmann, G. Kathagen, K. Moritz, R. Zeglin, and T. B˘ning. BeeHive–An Energy-Aware Scheduling and Routing Framework. Technical report-pg439, LSIII, School of Computer Science, University of Dortmund, 2004.

[220] H. F. Wedde, M. Farooq, C. Timm, J. Fischer, M. Kowalski, M. Langhans, N. Range, C. Schletter, R. Tarak, M. Tchatcheu, F. Volmering, S. Werner, and K. Wang. BeeAdHoc–An Efficient, Secure, Scalable Routing Framework for Mobile AdHoc Networks. Technical report-pg460, LSIII, School of Computer Science, University of Dortmund, 2005.

[221] H. F. Wedde, M. Farooq, and Y. Zhang. BeeHive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior. In *Ant Colony Optimization and Swarm Intelligence, LNCS 3172*, pages 83–94. Springer Verlag, Sept 2004.

[222] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler. *The Linux Networking Architecture*. Prentice Press, 2004.

[223] M. Weiser. The computer for the 21st century. *Scientific American*, pages 933–940, 1991.

[224] M. Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, 1993.

[225] M. Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):74–84, 1993.

[226] M. Weiser. The world is not a desktop. *Interactions*, 1(1):7–8, 1994.

[227] G. Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, San Francisco, CA, 1999.

[228] A. R. P. White. *SynthECA: A Synthetic Ecology of Chemical Agents*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, August 2000.

[229] T. White. Routing with swarm intelligence. Technical report sce-97-15, Systems and Computer Engineering Department, Carleton University, Canada, 1997.

[230] T. White and B. Pagurek. Towards multi-swarm problem solving in networks. In *3rd International Conference on Multi-Agent Systems (ICMAS 1998), 3-7 July 1998, Paris, France*, pages 333–340. IEEE Computer Society, 1998.

[231] T. White and B. Pagurek. Application oriented routing with biologically-inspired agents. In *Proceedings of Genetic Evolutionary Computation Conference (GECCO)*, pages 1453–1454, San Francisco, CA, USA, July 1999. Orlando, Florida, Morgan Kaufmann Publishers Inc.

[232] T. White and B. Pagurek. Emergent behaviour and mobile agents. In *Proceedings of the workshop on Mobile Agents Coordination Cooperation Autonomous Agents*, Washington, Seattle, May 1-5 1999.

[233] T. White, B. Pagurek, and D. Deugo. Biologically-inspired agents for priority routing in networks. In S. M. Haller and G. Simmons, editors, *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, May 14-16, 2002, Pensacola Beach, Florida, USA*, pages 282–287. AAAI Press, 2002.

[234] T. White, B. Pagurek, and F. Oppacher. ASGA: Improving the ant system by integration with genetic algorithms. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 610–617, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[235] T. White, B. Pagurek, and F. Oppacher. Connection management using adaptive agents. In *Proceedings of 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pages 802–809. CSREA Press, 1998.

[236] L. G. Williams and C. U. Smith. Pasa(sm): An architectural approach to fixing software performance problems. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*, pages 307–320. Computer Measurement Group, 2002.

[237] M. Woodside. Scalability metrics and analysis of mobile agent systems. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, pages 234–245, London, UK, 2001. Springer-Verlag.

[238] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 8-9, 1994, Proceedings*, volume 890 of *Lecture Notes in Computer Science*, pages 1–39. Springer, 1995.

[239] M. Wooldridge, N. R. Jennings, and D. Kinny. A methodology for agent-oriented analysis and design. In *ACM Third International Conference on Autonomous Agents*, pages 69–76, 1999.

[240] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison Wesley, 1995.

[241] Y. Yang, A. N. Zincir-Heywood, M. I. Heywood, and S. Srinivas. Agent-based Routing Algorithms on a LAN. In *IEEE Canadian Conference on Electrical & Computer Engineering*, 1442-1447 2002.

[242] J. Yu. RFC 2791: Scalable routing design principles, July 2000.

[243] L. Zhang, S. Deering, and D. Estrin. RSVP: A new resource reservation protocol. *IEEE Communications Magazine*, 31(9):8–18, 1993.

[244] Y. Zhang. Design and implementation of bee agents based algorithm for routing in high speed, adaptive and fault-tolerant networks. Master thesis, LSIII, The University of Dortmund, Germany, 2004.

[245] W. Zhong and D. Evans. When ants attack: Security issues for stigmergic systems. UVA CS Technical Report, CS-2002-23, Department of Computer Science, University of Virginia, April 2002.

[246] H. Zhu. Formal specification of agent behaviour through environment scenarios. In *Formal Approaches to Agent-Based Systems, First International Workshop, FAABS 2000 Greenbelt, MD, USA, April 5-7, 2000, Revised Papers*, volume 1871 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2000.

[247] H. Zhu. Slabs: A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(5):529–558, 2001.