University of Dortmund
Department of Computer Science
Project Group 460

BeeAdHoc: an Efficient, Secure and Scalable Routing Framework for Mobile AdHoc Networks

Final Report

Instructors:

Prof. Dr. Horst F. Wedde (wedde@ls3.cs.uni-dortmund.de)

ME Muddassar Farooq (muddassar.farooq@ls3.cs.uni-dortmund.de)

Project Manager:

ME Muddassar Farooq (muddassar.farooq@ls3.cs.uni-dortmund.de)

Students:

Julia Fischer (wjfischer@gmx.de)

Martin Kowalski (martin.kowalski@gmx.de)

Michael Langhans (michael.langhans@udo.edu)

Niko Range (niko.range@uni-dortmund.de)

Cornelia Schletter (schletter@debitel.net)

Recai Tarak (recai@genion.de)

Michel Tchatcheu (stchatcheu@yahoo.fr)

Constantin Timm (constantin.timm@uni-dortmund.de)

Friedrich Volmering (friedrich.volmering@gmx.de)

Sebastian Werner (mail@kryptoco.de)

Kai Wang (kaiwang@web.de)

Contents

1. Bees in Nature **1**

1.1. The colonies behaviour . . . 1

1.2. Communication by the means of dances . . . 2

1.2.1. The waggle dance . . . 2

1.2.2. The tremble dance . . . 3

1.3. Adaptability and the bees environment . . . 6

2. Agent Model of Bees **11**

2.1. Forager . . . 11

2.2. Recruits and bees at hive . . . 13

2.3. Scouts . . . 14

3. Implementation of BeeAdHoc in ns-2 **15**

3.1. Introduction . . . 15

3.1.1. Ns-2 . . . 15

3.2. OTcl . . . 16

3.3. Scenario generation . . . 16

3.4. BeeAdHoc implementation . . . 18

3.5. Implementation of Swarm cocepts in ns-2 . . . 19

3.5.1. Swarms Model in BeeAdHoc . . . 19

3.5.2. Swarms in ns-2 . . . 20

4. Mobile Agent in BeeAdHoc **22**

4.1. Forager . . . 22

4.2. Packer and Swarms . . . 26

4.2.1. Packer . . . 26

4.2.2. Swarms . . . 28

4.3. Scouts . . . 31

4.4. Checking the model by Simulation . . . 32
4.4.1. Forager . . . 32
4.4.2. Swarms . . . 35

5. Simulation Results with Transport Protocol TCP **38**
5.1. Changes in NS2 . . . 38
5.2. Simulation Runs . . . 38
5.3. Testing Mobility Behaviour . . . 39
5.3.1. Node Velocity . . . 39
5.3.2. Pause Time . . . 42
5.4. Testing Send Rate Behaviour . . . 42
5.5. Testing Packet Size Behaviour . . . 44
5.6. Testing the Influence of Increasing Number of Nodes . . . 48
5.7. Summary . . . 48

6. Scalability Model Of BeeAdHoc **52**
6.1. Introduction . . . 52
6.2. Motivation . . . 52
6.3. Getting BeeAdHoc Scalable:Idea . . . 52
6.3.1. Starting Point . . . 52
6.3.2. The Idea . . . 53
Introduction . . . 53
Scouting . . . 54
Forager Passing . . . 55
6.4. BeeAdHoc Scalable: Model . . . 55
6.4.1. Definitions . . . 55
6.4.2. Introduction . . . 56
6.4.3. Information Carried By Scouts . . . 56
6.4.4. Launching A Scout . . . 56
6.4.5. Forwarding A Scouting Scout . . . 57
6.4.6. A Scout At Its Destination . . . 57
6.4.7. Forwarding A Scout On Its Way Home . . . 57
6.4.8. A Scout At Its Home . . . 58
6.4.9. Information Carried By Foragers . . . 58
6.4.10. Initiation A Forager . . . 58
6.4.11. Forwarding A Forager . . . 58
6.4.12. A Forager At Its Destination . . . 59

6.5. Getting BeeAdHoc Scalable: Further Ideas . . . 59
6.6. Implementation Of The Scalability Model In NS2 . . . 59
6.6.1. Introduction . . . 59
6.6.2. Scouting . . . 59
6.6.3. Scout Arrival: Destination/Source . . . 63
6.6.4. Scout On Way Back . . . 64
6.6.5. Forager Passing . . . 65
6.6.6. Next Hop List . . . 66

7. Scalability Issues in MANETs **67**
7.1. Simulation framework . . . 67
7.1.1. Introduction . . . 67
7.1.2. Considered Algorithms . . . 67
AODV . . . 67
DSR . . . 68
DSDV . . . 69
7.1.3. Metrics . . . 69
Energy per user data . . . 69
Success rate . . . 69
Delay . . . 69
Throughput . . . 70
7.1.4. Simulation environment . . . 70
Previous tests . . . 70
Several other tests . . . 70
7.1.5. Testing results . . . 73
7.2. Testing results . . . 73
7.2.1. simulation runs . . . 73
7.2.2. Testing Energy behavior per user data . . . 73
7.2.3. Testing Delay behavior . . . 74
7.2.4. Testing Throughput behavior . . . 76
7.2.5. Testing control packet behavior . . . 79
7.2.6. Testing the packet delivery ratio behavior . . . 81

8. Security in MANETs **83**
8.1. General attacks on Adhoc Networks . . . 84
8.2. Cryptographic Routing Algorithms . . . 88
8.2.1. AODV . . . 88

8.2.2. DSR 89
8.3. Typs of Cryptography 89
8.3.1. Asymmetric Cryptography 89
8.3.2. Symmetric Cryptography 90
8.4. Secure Cryptographic Routing Algorithms 90
8.4.1. SAODV 90
8.4.2. ARIADNE 93

9. Artificial Immune Systems 94
9.1. The Human Immune System 95
9.2. How the Immune System protects the body 96
9.2.1. The Immune Cells 98
9.2.2. Reinforcement Learning and Immune Memory 99
9.2.3. Pattern Recognition 100
9.2.4. Immunologic Self/Nonself Discrimination 100
9.2.5. AIS Definitions 101
9.3. IDS 101
9.3.1. Inspired by Nature 102
9.3.2. AIS Framework 102

10. Measuring Energy Consumption 104
10.1. Profiling 104
10.1.1. Energy Model 104
10.1.2. Profiling Framework 105
10.2. NS-2 108
10.2.1. NS-2 Modifications 108
10.2.2. Testing Results 109

11. BeeGuard 117
11.1. Security Model 117
11.1.1. Examples of Non Relevant Attacks for BeeAdHoc 118
11.1.2. Relevant Attacks for BeeAdHoc 119
11.2. NS-2 Implementation of BeeGuard 121
11.2.1. BeeGuard::AntiBody 123
11.2.2. BeeGuard::HoneyPot 124
11.2.3. BeeGuard::ScoutFreeze 126
11.2.4. BeeGuard::TrafficPattern 127

11.2.5. Embedding BeeGuard in BeeAdhoc . . . 130
11.3. Evaluation of the Energy Performance of BeeGuard . . . 131
11.4. Testing the Functionality of BeeGuard . . . 132
11.4.1. BeeGuard::AntiBody . . . 132
11.4.2. BeeGuard::HoneyPot . . . 134
11.4.3. BeeGuard::ScoutFreeze . . . 136
11.4.4. BeeGuard::TrafficPattern . . . 138
11.5. Conclusion . . . 141

12. Routing Framework in Linux 143
12.1. Overview . . . 143
12.1.1. The Linux Netfilter Architecture . . . 143
12.1.2. Using the IP header for transporting bee data . . . 144
IP options . . . 145
Source routing . . . 145
The BeeAdhoc option . . . 146
12.1.3. Introduction to our implementation . . . 147
Netfilter FORWARD-chain . . . 147
Netfilter OUTPUT-chain . . . 147
Netfilter INPUT-chain . . . 148

13. BeeAdHoc in Linux 149
13.1. Kernel Diffs . . . 149
13.1.1. Necessary modifications inside the original Linux kernel code . . . 149
Modifications inside /include/linux/ip.h . . . 149
Modifications inside /net/ipv4/ip_options.c . . . 150
Adding and modifying ipv4options match . . . 150
13.1.2. Optional modifications inside the original Linux kernel code . . . 151
13.2. Kernel Module . . . 151
13.2.1. The kernel module . . . 151
13.2.2. The module core: ipt_BEEADHOC.c . . . 152
Concurrency . . . 152
Core initialisation . . . 153
The ipt_beeadhoc_target function . . . 153
Netfilter OUTPUT-chain . . . 154
Netfilter FORWARD-chain . . . 158
Netfilter INPUT-chain . . . 159

Using ACPI . . . 160
13.3. Datastructs . . . 162
Structures . . . 162
The BeeAdHoc-API . . . 163
Helper functions . . . 165
13.4. ProcStat . . . 170
13.5. Scouting inside the kernel . . . 173
13.6. Iptables . . . 180
Iptables BeeAdHoc-target . . . 180
Iptables IPV4OPTIONS-match . . . 180
Using iptables . . . 180
13.7. Swarms implementation . . . 181
13.7.1. Foreign and Own Forager . . . 182
13.7.2. Swarm triggering . . . 183
13.7.3. Build the swarm packet . . . 184
13.7.4. Sending and forwarding the swarm . . . 186
13.7.5. Swarm arrives at destination . . . 187

14. Performance Evaluation Framework for MANETs in Linux 188
14.1. Introduction . . . 188
14.2. Tested Algorithms . . . 188
14.2.1. AODV . . . 188
14.2.2. OLSR . . . 189
14.3. Testing environment and settings . . . 190
14.3.1. Hardware . . . 190
14.3.2. Software . . . 191
Minimal Linux Installation . . . 192
14.4. Scripts . . . 192
14.4.1. Motivation . . . 193
14.4.2. Design of the testing framework . . . 194
14.4.3. APE . . . 198
14.4.4. Cloning . . . 199
14.5. Scenarios . . . 200
14.5.1. UML . . . 200
14.5.2. Quasireality . . . 203
14.5.3. Reality . . . 205

14.6. UML . . . 207
14.6.1. Description . . . 207
14.6.2. Modifications . . . 208
14.6.3. Results . . . 210
14.6.4. TCP Results . . . 211
14.6.5. UDP Results . . . 211
14.7. Quasireality . . . 212
14.7.1. Results . . . 212
14.7.2. TCP Results . . . 214
14.7.3. UDP Results . . . 218
14.7.4. Analysis . . . 219
14.7.5. Energy consumption . . . 219
14.8. Reality . . . 220
14.8.1. mobility tests . . . 220
14.8.2. static tests . . . 220
14.9. Analysis . . . 220

A. Appendix **222**
A.1. UML . . . 222
A.1.1. Filesystem . . . 222
A.1.2. Network . . . 223
A.1.3. Starting User-mode Linux . . . 223
A.1.4. Dealing with problems . . . 224
A.2. Thrulay . . . 224
A.3. Netperf . . . 225
A.4. FTP . . . 226
A.5. Wget . . . 226
A.6. Ping . . . 226
A.7. Traceroute . . . 227
A.8. TCPDump . . . 227
A.9. Scenario- Editor . . . 228
A.9.1. Standard Run . . . 228
A.9.2. Methods . . . 228
A.9.3. Data structures . . . 230
A.10.ITG . . . 231
A.11.STG . . . 232

A.12. Mackill . . . 233
A.13. WSCCS . . . 235
A.13.1. Syntax . . . 235
A.13.2. Semantics . . . 236
A.13.3. The probability workbench . . . 240

Bibliography **244**

List of Tables

5.1. TCP-Runs for BeeAdHocBMAS (x=100), BeeAdHoc (x=0), DSR (x=300), AODV (x=600) . . . 39

14.1. Detailed node overview . . . 190
14.2. static node placement . . . 206
14.3. UML - Thrulay - Throughput(Mbit/s) . . . 211
14.4. UML - Netperf - 900sec - Throughput(Mbit/s) . . . 211
14.5. UML - Thrulay - 900sec - average RTT . . . 211
14.6. UML - Thrulay - 900sec - Control packets . . . 212
14.7. UML - sqtg - 300sec - static - 3 hops . . . 212
14.8. Quasireality - Thrulay - 120sec - Throughput . . . 215
14.9. Quasireality - Thrulay - 120sec - avg RTT . . . 215
14.10Quasireality - Thrulay - 120sec - control-packets . . . 215
14.11Quasireality - 900sec - Throughput . . . 215
14.12Quasireality - 900sec - avg delay . . . 216
14.13Quasireality - 900sec - control-packets . . . 216
14.14Quasireality - 900sec -total data packets delivered . . . 216
14.15Quasireality - D-ITG - 300sec - static - 3 hops . . . 218
14.16Quasireality - sqtg - 300sec - static - 3 hops . . . 218
14.17Quasireality - D-ITG - 300sec - VoIP . . . 218
14.18Joule consumption during 900sec testrun . . . 220
14.19Reality - Thrulay - 720sec - mobility . . . 220
14.20Reality - Thrulay - 720sec - static . . . 220

List of Figures

1.1. dancing after long seek times . . . 4
1.2. influence of the profitability of the food source on the *tremble dance* . . . 5
1.3. The location of tremble and waggle dances inside a hive . . . 6
1.4. how the tremble dance calms congestions in nectar unloading . . . 7
1.5. how a forager assesses the quality of her food source . . . 9
1.6. how the food source quality influences the colonies behaviour . . . 10

2.1. Bienen MAS: Variablen . . . 14

4.1. WSCCS model: variables . . . 27

5.1. Energy results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x) 40
5.2. Hopcount results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x) . . . 40
5.3. Delay results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x) 41
5.4. Throughput results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x) . . . 41
5.5. Delay results depending on pause time (from runs 27+x, 28+x, 29+x) . . 42
5.6. Throughput results depending on pause time (from runs 27+x, 28+x, 29+x) 43
5.7. Energy results depending on pause time (from runs 27+x, 28+x, 29+x) . 43
5.8. Throughput results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x) . . . 44
5.9. Energy results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x) 45
5.10. Delay results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x) 45
5.11. Control packets results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x) . . . 46
5.12. Energy results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x) . . . 46

5.13. Delay results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x) 47
5.14. Throughput results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x) 47
5.15. Control packets results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x) 48
5.16. Energy results depending on node number (from runs 9+x, 13+x, 14+x, 29+x) 49
5.17. Delay results depending on node number (from runs 9+x, 13+x, 14+x, 29+x) 49
5.18. Throughput results depending on node number (from runs 9+x, 13+x, 14+x, 29+x) 50
5.19. Control packets results depending on node number (from runs 9+x, 13+x, 14+x, 29+x) 50

6.1. Situation Before BeeAdHoc 53
6.2. Choice Of rHops 54
6.3. BeeAdHoc: Forager Passing 60
6.4. BeeAdHoc: Scout Handling 61

7.1. Overview of the tests in node 50 71
7.2. example of a shell script 71
7.3. Run configuration 73
7.4. Energy depending on pause time from runs x+1, x+2, x+3, x+4 x+5 . . 74
7.5. Energy depending on number of nodes average values from runs x+1, x+16, x+31, x+46, x+61, x+76 75
7.6. Energy depending on node velocity average values of all runs 75
7.7. Energy depending on node velocity just for AODV, BeeAdHoc and DSDV 75
7.8. Delay depending on pause time from runs x+6, x+7, x+8, x+9 x+10 . . . 76
7.9. Delay depending on number of nodes average values from runs x+5, x+20, x+35, x+50, x+65, x+80 77
7.10. Delay depending on node velocity average values of all runs 77
7.11. Throughput depending on pause time from runs x+6, x+7, x+8, x+9 x+10 78
7.12. Throughput depending on number of nodes average values from runs x+2, x+17, x+33, x+48, x+63, x+78 78
7.13. Throughput depending on node velocity average values of all runs 79
7.14. control packet depending on pause time from runs x+1, x+2, x+3, x+4 x+5 80

7.15. Control packet depending on number of nodes average values from runs x+11,x+26,x+41,x+56,x+71,x+86 80
7.16. control packet depending on node velocity average values of all runs . . . 80
7.17. control packet depending on node velocity average values of all runs . . . 80
7.18. Packet delivery ratio depending on pause time from runs x+51, x+52, x+53, x+54 x+55 81
7.19. Packet delivery ratio on number of nodes average values from runs x+4, x+19, x+34, x+49, x+64, x+79 82
7.20. Packet delivery ratio depending on node velocity average values of all runs 82

8.1. Asymmetric Cryptography: Signing 90
8.2. SAODV: Route single signature extension [Zap01] 91

9.1. The Anatomy of the Human Immune System[Imm] 97
9.2. How the immune system defends the body [dCvZ99] 98
9.3. Structural division of the cells and secretions of the immune systems [dCvZ99] 99
9.4. Components of the immune system that satisfy the requirements for the development of an efficient intrusion detection system (IDS) [dCvZ00] . . 102
9.5. Relationship between the network intrusion detection systems and the immune system[dCvZ00] 103

10.1. Complete Cyclecounts 111
10.2. Network Traffic Energy Consumption 112
10.3. Cycle Allocation: Pausetime 1 113
10.4. Cycle Allocation: Pausetime 30 114
10.5. Cycle Allocation: Pausetime 60 115

11.1. BeeGuard : Theoretical Model 122
11.2. Cyclecounts BeeAdHoc vs BeeGuard 131
11.3. Cyclecounts for Antibodies 134
11.4. Testing Antibodies: Controlpackets 135
11.5. Batterystats for Honeypot Tests 137
11.6. Packets send by attacking Node 137
11.7. Cyclecount for Honeypot 138
11.8. Cyclecount for Trafficpattern 141
11.9. Share of Corrupted Datapackets 141

12.1. Implementation overview . . . 143
12.2. Netfilter hooks . . . 144
12.3. Standard conform IP header . . . 144
12.4. IP options . . . 145
12.5. routing table . . . 146
12.6. Source route option . . . 146
12.7. BeeAdhoc option . . . 147

13.1. sk_buff . . . 154
13.2. Configuring netfilter for our needs . . . 181
13.3. Configuring netfilter for our needs (without kernelpatch) . . . 182

14.1. real topology . . . 193
14.2. topology: everybody sees everybody . . . 193
14.3. Performance Evaluation Framework . . . 194
14.4. UML topologies (part1) . . . 201
14.5. UML topologies (part2) . . . 202
14.6. quasireality topologies (part1) . . . 204
14.7. quasireality topologies (part2) . . . 205
14.8. quasireality test setup . . . 205
14.9. reality test szenario scheme . . . 207
14.10reality tests . . . 208
14.11example UML network . . . 209
14.12graphical interpretation of UML tests - part 1 . . . 213
14.13graphical interpretation of UML tests - part 2 . . . 214
14.14Quasireality - 900sec - control-packets . . . 217
14.15Quasireality - 900sec - avg delay . . . 217
14.16Quasireality - 900sec - throughput . . . 217
14.17Quasireality - 900sec - jitter . . . 217
14.18Quasireality - 900sec - average standard deviation . . . 217
14.19Quasireality - 900sec - total data packets delivered . . . 217
14.20Energy consumption . . . 219

A.1. 'thrulay' tcp test . . . 225
A.2. 'thrulay' udp test . . . 225
A.3. 'netperf' tcp test . . . 225
A.4. tcpdump example output . . . 227

A.5. The output file of the . . . 229
A.6. Scenario editor and abfrage . . . 230

Listings

6.1. Scouting Phase . . . 62
6.2. Scout: Source Destination Arrival . . . 63
6.3. Scout On Way Home . . . 64
6.4. Forager Passing . . . 65
6.5. Entrance::saveRoute() . . . 66
10.1. Profiling in NS-2 Methods . . . 108
11.1. AntiBody Constants and Datatypes . . . 123
11.2. AntiBody Methods . . . 124
11.3. Immunity Decay Function . . . 124
11.4. HoneyPot Constants and Datatypes . . . 125
11.5. HoneyPot Methods . . . 125
11.6. ScoutFreeze Methods . . . 126
11.7. TrafficPattern Constants and Datatypes . . . 127
11.8. TrafficPattern Methods . . . 128
11.9. Calculating Route IDs . . . 128
11.10Additional Beedefinitons . . . 130
11.11Antibody Failureinjection . . . 132
11.12HoneyPot Failureinjection . . . 136
11.13Trafficpattern Failureinjection 1 . . . 139
11.14Trafficpattern Failureinjection 2 . . . 139
13.1. Changes in ip.h . . . 149
13.2. Changes in ipoptions.c . . . 150
13.3. Output chain code . . . 154
13.4. QueueTaskletFunction - Read the queue and build packets . . . 155
13.5. creating space for IP options inside an sk_buff . . . 157
13.6. Inserting IP options . . . 157
13.7. Energy check function . . . 159
13.8. ACPI initialisation . . . 160

13.9. Battery watcher kernel thread 161
13.10Main hash table. 162
13.11daddr_list_entry struct 162
13.12Forager struct 162
13.13Source route struct 163
13.14beeadhoc_struct_gc 165
13.15forager_rate_lifetime 166
13.16insert_forager_into_array 167
13.17get_opt_forager 169
13.18ipt_BeeAdHoc_stat.c 170
13.19procfs output 171
13.20cycle count 172
13.21scout packet 174
13.22retransmit entry 174
13.23scout seen array entry 175
13.24scout send tasklet 175
13.25Copy scout packet and add timer 176
13.26scout handling 177
13.27Information evaluating 178
13.28creating a forager 179
13.29daddr_list_entry struct 182
13.30swarm send condition 183
13.31swarm packet struct 184
13.32send_forager struct 184
13.33buildswarm function 184
13.34swarm arrival 186
13.35put forager from swarms 187
14.1. topologychange script 195
14.2. sample scenario file 196
14.3. sample topology 197
14.4. batlog script 197
A.1. Changing process_backlog 234

1. Bees in Nature

> A honey bee colony can skillfully choose among nectar sources.
>
> It will selectively exploit the most profitable source in an array and will rapidly shift its foraging efforts following changes in the array.[SCS91]

Surprisingly this ability of a honey bee colony comes from a rather simple behaviour of each individual bee that needs little communication. The individual bee just uses local information to make decisions in a decentralized fashion without any central control.

1.1. The colonies behaviour

There are several kinds of worker bees in a honey bee colony:

* food-storer
* scout
* forager

The food collection is organized by the colony by recruiting bees for different jobs. The recruitment is managed by the forager bees which can perform dances to communicate with their fellow bees inside the hive and recruit them. At the entrance of the hive is an area called the dance-floor, where dancing takes place.[1]
A colony only has a few scouts who become foragers and then recruit foragers after discovering a proper food source.[Fri65]
A forager which wants to advertise her food source in order to recruit more bees for the food source performs the waggle dance at the dance-floor. If she experiences a longer wait time to find a food-storer bee then she does a *tremble dance*. The tremble dance is an indication to other foragers to stop dancing because the rate of collecting nectar is

[1] Therefore Seeley says in [ST92]:

> (the dance-floor is) heart of the decision making process

greater than the rate of processing it.
When a food source no longer gives any profit, the foragers[2] will soon abandon this source, visit the dance-floor and be recruited for another source or stay at hive. Sometimes a forager checks if her former source has recovered. If she is successful, other bees of her former collecting group[3] return to the source on their own initiatives.[Fri65]

1.2. Communication by the means of dances

In order to find out about the meaning of different dances and how other bees understand the message of a dance, biologists have done many experiments. Their results are going to be presented in the following.[4]

1.2.1. The waggle dance

The waggle dance is an advertisement for the food source of the dancer. Another forager can leave her food source and watch out for a well advertised food source.[5] What kind of information is offered by the advertisement and what is the criteria for choosing an advertised food source ?

The observations made are quite unexpected.[6]. A forager randomly follows dances of multiple recruiting foragers and seems to response randomly as well. Especially she *does* not *compare several* dances.
Where does this claim come from and how is this behaviour revealed in the experiments? A dance does not seem to contain any information that helps to choose a food source.[7]

[2] Each forager has her own area in the flower patch where she returns every time. So the foragers do not get in trouble with each other.[Fri65]
[3] *A collecting group* is a number of foragers who fly to the same patch of flowers.
[4] T.D.Seeley quotes:

...the key to understanding the message of any communicatory behaviour is knowing what the sender consistently experiences before performing the behaviour.

[5] In the section *Adaptability and the bees environment* is explained, what makes her leaving or dancing.
[6] *Seeley* and *Towne* at first suppose the opposite as a conclusion from earlier results of their work:

* the dances are performed at the same time close to each other
* foragers watch several dances before leaving the hive
* the quality of the food source influences the dancing behaviour

[7] A piece of information which may be coded in a dance is the quality of a food source, because a dance lasts longer if the food source is more profitable. Nevertheless the duration of a dance is not the precise information since it varies broadly for the same source. The volume of the dance does not code the quality of the food source, either. It does not change although the quality does. One possibility still remains but is not investigated by *Seeley* and *Towne* in their experiments. That is

If the behaviour is random, a richer food source only can be preferred if there are *more* advertisements for it. If the quantity of dances and their duration is greater than the probability another forager watches a dance for this source will increase. This can be observed in the following result of an experiment done by *Seeley* and *Towne*:

date	food source	concentration of sugar solution (Mol/L)	observed/expected recruits[8]
5th July	north	2.5	37 / 36.2
	south	2.0	24 / 24.8
6th July	north	2.0	39 / 34.3
	south	2.5	57 / 61.7
...			

1.2.2. The tremble dance

[9]

> While they run about the combs in an irregular manner and with a slow tempo, their bodies, as a result of quivering movements of the legs, constantly make a trembling movement forward and backward, and right and left. During this process they move about on four legs, with the forelegs, themselves trembling and shaking, held aloft approximately in the position in which a begging dog holds its forepaws. ...[Fri65] (translated by T.D.Seeley)

There is an important hint to the meaning of the message of the *tremble dance*: Foragers are more likely to perform the tremble dance if they have to wait long for a *food-storer* bee to unload their nectar after their arrival at hive. This observation comes from one of *Seeley´s* experiments in which he summarizes the message of the tremble dance as follows:

> I (a bee) have visited a rich nectar source worthy of greater exploitation, but already we have more nectar coming into the hive than we can handle.

whether a bee codes the quality of the food source on her own.
This idea is supported by a remark of *Von Frisch*, that the *distance is encoded* and bees refuse long distanced food sources and do not care about the profitability of those sources.[Fri65]
In the section *Adaptability and the bees environment* one can find how the *quality of the food source* is assessed and how it influences the dancing behaviour.

[8] [expected numbers of recruits] = [numbers of all recruits for both sources of this day] ∗ [probability, a bee is recruited by the waggle dance for the food source]
In the assumption this probability is equal to ratio of dances for this food source with regard to the dancing for all the sources.

[9] The results of this subsection are from [See92] but other sources may be quoted, too.

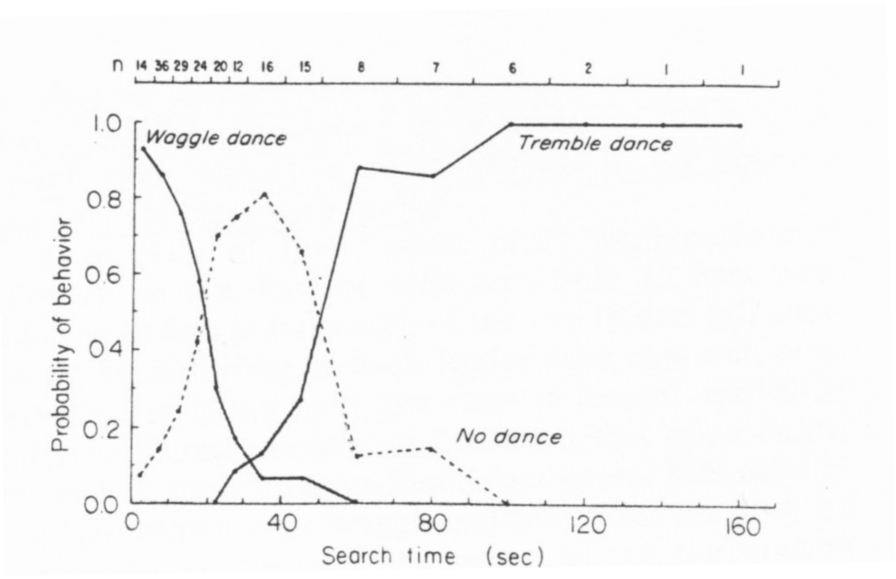


Figure 1.1.: In the area of middle seek time there is neither any waggle or tremble dance performed. That affirms the assumption that the two dances have opposite meaning. Fig. 7 from [See94]

In addition to a long seek time, the tremble dancer experiences, her food source need to have some profit.

In the plots of the experiments result one can see the relation between the probability to dance and these two factors.[10]

Which kind of bees are addressed by the *tremble dance* and what is their answer to its message ?

Foragers perform the tremble dance on the dance-floor and in the *brood nest* as well , whereas the waggle dance is limited to the dance-floor.[11] So maybe bees in the hive are addressed, too. According to *Seeley* worker bees in the hive are ordered by the tremble dancers to give up their jobs and to unload nectar.[12]

The experiment, which led Seeley to this assumption, is described in figure 1.4 on page 7.

Other foragers are told to stop their waggle dances, which is reported in the same

[10] See figure 1.1 on page 4 and figure 1.2 on page 5

[11] In figure 1.3 on page 6 it is shown, on which places in the hive the different dances are performed.

[12] Still some questions remain:

* Which kinds of workers are recruited to food-storer bees?
* Will *food-storer* bees who watch a tremble dance work with greater effort?

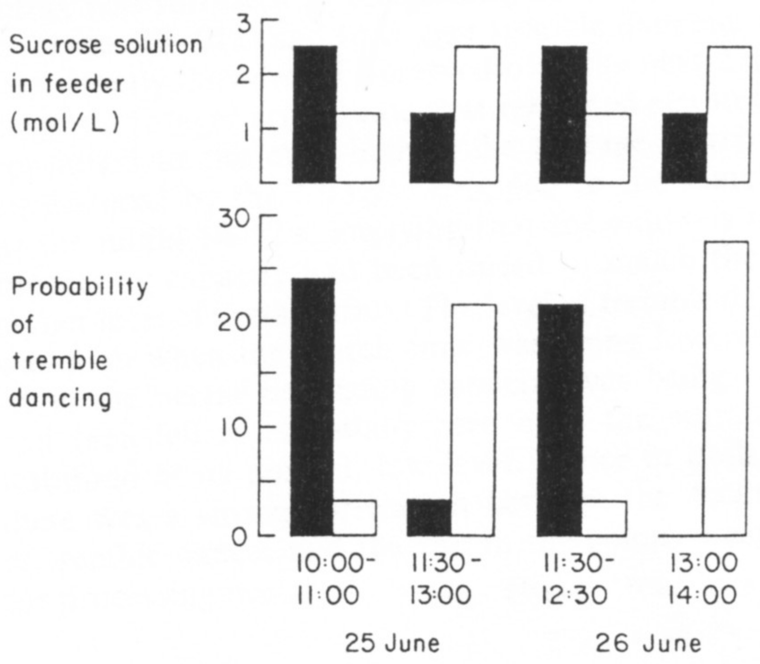


Figure 1.2.: The number of foragers have been hold on a high level, so that there is a lack of food-storer bees during the whole experiment. Fig. 8 from [See94]

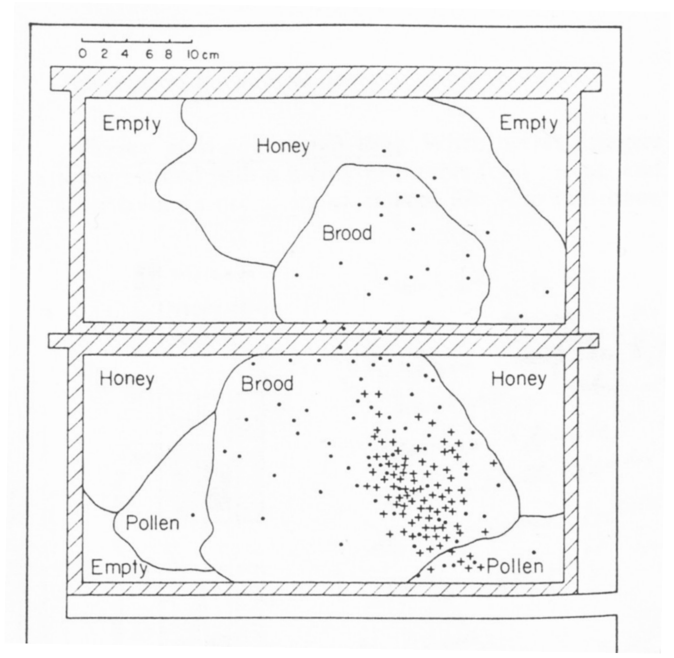


Figure 1.3.: The crosses which mark the location of waggle dances are clustered in an area called the dance-floor. In contrast, the tremble dance is distributed on the brood nest and on the dance-floor as well. Fig. 4 from [See94]

experiment as well.

1.3. Adaptability and the bees environment

Some forms of adaptability already appeared in the section *communication by the means of dances*. For example, a bee can abandon her food source and follow the advertisement to another one later on.
This advertisement, communicated by a *waggle dancer*, can be viewed in terms of adaptability. And by *tremble dancing* a forager adapts to congestions in the nectar unloading.

* Which conditions of the environment are the foragers adapted to?
* Are foragers exchanging information with others for that purpose?
* How is the adaptability revealed in the bees behaviour?
* How does this influence the colonies situation?

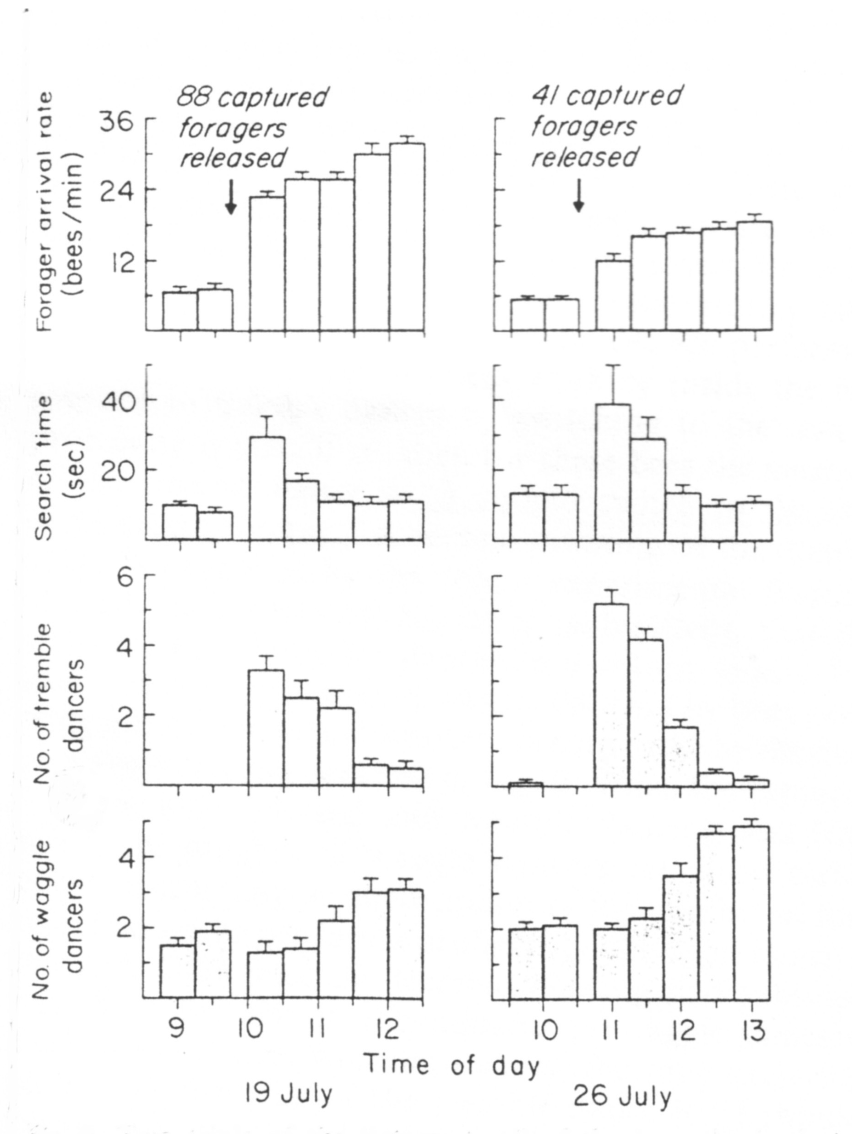


Figure 1.4.: A great number of forager are freed to return to hive at the same time, so that a *congestion in nectar unloading* occurs. After tremble dancing the seek time decreases on its old level. So one can suppose that new *food-storer* bees have been recruited.
Although there is a sudden increase in the number of foragers the total number of *waggle dances* is unchanged. So the individual must dance less. Fig. 9 from [See94]

A forager assesses the quality of her food source without communication and without comparing it directly with other food sources:
In a experiment it gets obvious that long distanced sources get a worse rating even if they had a higher sugar concentration:

distance(m):	50	1250	
sugar concentration(mol/L):	0.75	1.00	2.50
probability of dancing:	0.5	0.1	0.73

Therefore other bees in the hive are not able to rate the food source since they do not know about its distance. Usually a bee just visits one food source and so cannot compare sources directly.

The assessment of a food source depends not only on the sugar concentration but also on the distance of the food source and the season.
In the *summer* an experiment with a bee colony has been made in order to find out how a forager measures her profit of one flight [See94]. In the summer a forager is interested in the *efficiency*[13] but not in the total gain of energy[14] or the rate of energy-gain[15]. In the *autumn* the *total gain of energy* is more important for the colony facing the coming winter.
There are advantages of measuring the total gain as well if the *colony is small*[16] or there are little supplies.[17]
A forager simply translates the quality of her source into her dance, which is shown in figure 1.5 on page 9.

If the quality changes the duration of her waggle dance changes proportionally, too. With an individual *threshold* a forager decides if she will dance or not. The threshold is useful for adaptability to the *food supplies of the colonies environment*: A forager raises her threshold if food is plenty and lowers it if food is rare.

In the section *Communication by the means of dances* one can see that the duration of the *waggle dance* has an impact on the numbers of bees being recruited for the advertised food source. So a single forager and her adaptability to environment play a role in the life and organisation of the whole colony. This is not only with regard to her dancing behaviour but also to her decisions to abandon a food source and being recruited herself.

[13] [total gain of energy]/[costs]
[14] [gain of energy]−[costs]
[15] [total gain of energy]/[time]
[16] than it seems more reasonable for the colony to grow fast
[17] but this is just supposed in [See94] and not investigated in a experiment any longer

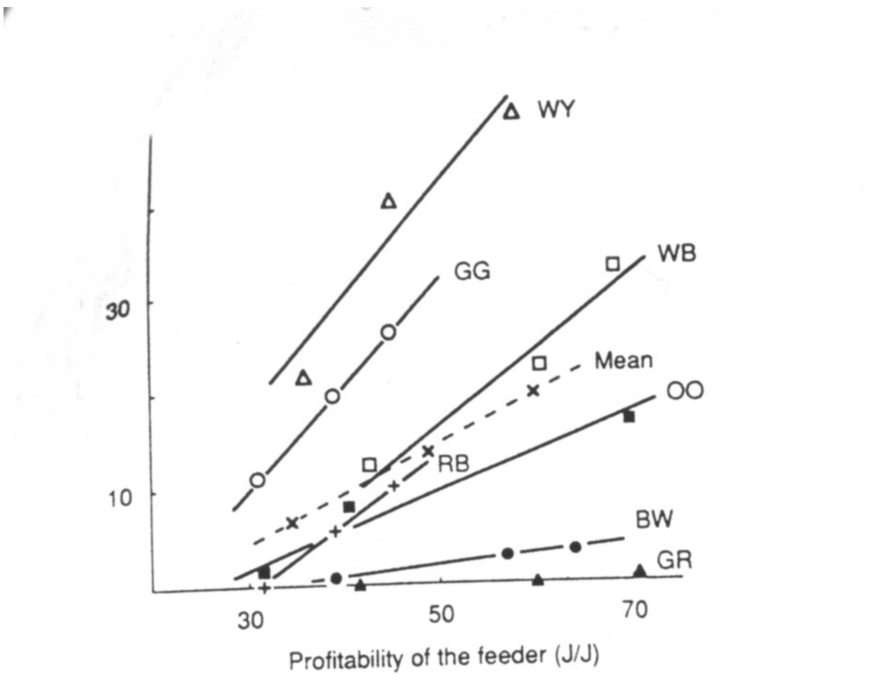


Figure 1.5.: The individual bees, WY, GG . . . GR each assess the quality of her food source considering its *efficiency*. Each of them has an individual *threshold*. If the quality is below the threshold a forager does not dance the *waggle dance*. Fig. 1 from [CS93]

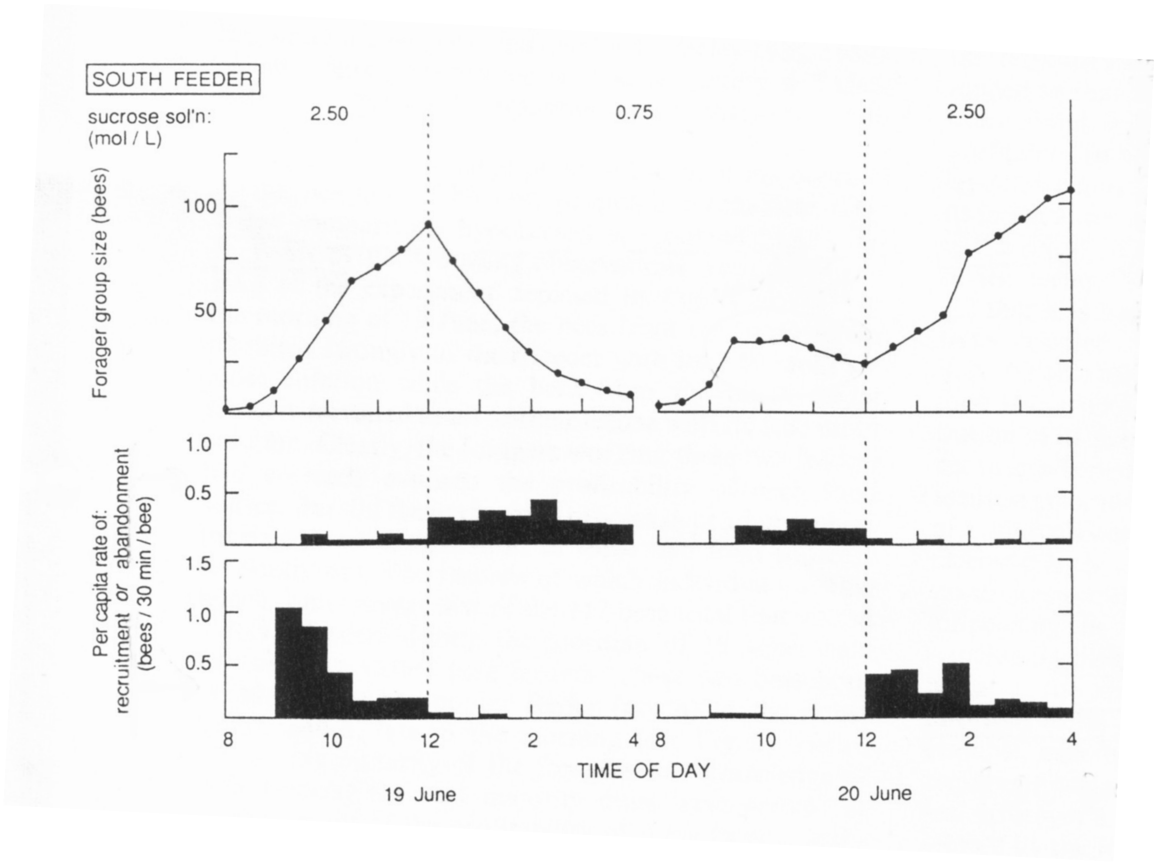


Figure 1.6.: *forager group size* is the number of individuals[19], which have been observed at the food source in the past 30 min. On 20th June some foragers still visited her food source like the day before, to check the source, but did abandon it rather quickly. Fig. 2 from [ST92]

How this simple behaviour of many bees makes the colony focus on richer food supplies, is shown in figure 1.6 on page 10.[18]

"'The *nectar supply* of flowers *frequently varies* in quantity and quality during daytime."'[Fri65] Therefore, a colony needs to cope with a sudden flood of nectar. This is being accomplished by *tremble dancers*,which is shown in figure 1.4 on page 7.

[18] A foragers behaviour of dancing seems not to be designed for informing her colony quickly if the supplies of her food source changes. This ability - which is called *sensory adaptability* - means, that a forager would dance vividly if the quality of her food source rises, but her interest would be decreasing and so would be her dances if the quality does not change anymore.

2. Agent Model of Bees

Is the behaviour of an individual bee described in chapter 1 sufficient to explain the behaviour of a colony in gaining food efficiently or are important aspects missing?
This question is of great importance, since we want to develop an algorithm which takes the benefits of this behaviour. Unfortunately, the model presented in this chapter is incomplete in terms of the bees behaviour described in chapter 1. Though many of the ideas included in the model reappear in the MAS of BeeAdHoc. [1]

2.1. Forager

WSCCS model

The waggle dance A forager, which flies to *source* *s*, tries to advertise others – *recruits* – to follow her by the waggle dance:

$$Recruit_s \equiv \omega : \overline{dance}.ShowSite_s + 1 : \surd.DontShow_s \tag{2.1}$$

If she does not succeed, she flies without recruiting:

$$DontShow_s \equiv 1 : \surd.Go_{s,G_s} \tag{2.2}$$

In the appendix one can look up the communication of agents in WSCCS[2] for explanation. On WSCCS the agents communicate by the action *dance*. The priorities ω in WSCCS help to express that a forager will recruit a packer whenever possible.
The recruited agent is informed by the recruiting forager where source s is located, afterwards:

$$ShowSite_s \equiv 1 : \overline{show_s}.Go_{s,G_s} \tag{2.3}$$

[1] The modelling of the bees is from the paper [Sum00]. There one can find how the model has been developed. The variables and actions are explained in table 2.1 on page 14
[2] Weighted Synchronous Calculus of Communicating Systems

The two communicate by $show_s$.
Go_{s,G_s} describes the flight to the food source.

The flight to the food source A forager needs at least G_s time steps to reach the source, but the arrival may be delayed with a probability of p_j. The numbers of foragers which visit a food source is rising with no limit unless this food source has a capacity. If its capacity is exhausted then the incoming foragers could not collect nectar from this food source. This is modelled by the following agent:

$$Site_s \equiv \sum_{i=1}^{C_s} 1 : \overline{pick_s}^{i}.Site_s + 1 : \surd.Site_s \tag{2.4}$$

The capacity is limited by C_s ; forager can collect nectar at source s by the action $pick_s$. The flight itself is described by the following agent:

$$Go_{s,\tau} \equiv \begin{array}{ll} p_j : \surd.\surd.Go_{s,G_s} + (1-p_j) : \surd.\surd.Go_{s,G_s-1} & \tau = G_s \\ \omega : pick_s.\surd.Succeed_{s,R_s} + 1 : \surd.\surd.Fail_{s,R_s} & \tau = 1 \\ 1 : \surd.\surd.Go_{s,\tau-1} & otherwise \end{array} \tag{2.5}$$

After the departure from hive ($\tau = G_s$) the flight can be delayed by $p_j : \surd.\surd.Go_{s,G_s}$. On arrival at the food source ($\tau = 1$) a forager attempts to pick some nectar($\omega : pick_s$) and returns to hive with a load of nectar($Succeed_{s,R_s}$). If she could not collect nectar she returns unsuccessfully($Fail_{s,R_s}$).

Arrival at hive A forager, which returns to hive to get her nectar unloaded, can stay there with a probability of p_r (*Other*) or can dance with a probability of p_s, which depends on the richness of the food source:

$$Succeed_{s,\tau} \equiv \begin{array}{ll} p_r : \surd.\surd.Other + (1-p_r)p_s : \surd.\surd.Recruit_s & \\ +(1-p_r)(1-p_s) : \surd.\surd.Go_{s,G_s} & \tau = 1 \\ 1 : \surd.\surd.Succeeds, \tau - 1 & otherwise \end{array} \tag{2.6}$$

If a forager is not successful in collecting nectar she always watches for another task in hive:

$$Fail_{s,\tau} \equiv \begin{array}{ll} 1 : \surd.\surd.Other & \tau = 1 \\ 1 : \surd.\surd.Fail_{s,\tau-1} & otherwise \end{array} \tag{2.7}$$

Comparison between real bees and the MAS In the model p_s expresses the probability to dance. The foragers evaluate the quality of their food sources without communicating

with other bees and with local information only; the evaluation is not included in the model but is fixed and not different between the individual bees. A forager's dance just lasts one time step and just advertises it to single bee or none.
The model does not include the ability of foragers to get an idea of the total nectar collecting rate of the colony by experiencing short or long times in search for a food-storer bee. So the adaptability to changing collecting rates is not modelled.

2.2. Recruits and bees at hive

WSCCS model Bees can be employed into different jobs at hive: Either they let themselves be recruited to a forager(*OnDFloor*), search for new food sources as a (*Scout*) or something different (*Other*).

$$OnDFloor \equiv \omega.dance.Follow + 1 : \surd.NoSite \tag{2.8}$$

($\omega.dance.Follow$) describes bees, which follow foragers on the dance-floor in order to be recruited. If they have not met a dancing bee, they may fly out to scout , stay on the dance-floor or look for other tasks in the hive:

$$NoSite \equiv p_o : \surd.Other + p_t : \surd.Scout + (1 - p_o - p_t) : \surd.OnDFloor \tag{2.9}$$

Recruits which follow a dancing forager are shown the path towards the food source in the following way:

$$Follow \equiv \sum_{s=1}^{n_s} 1 : show_s.Find_{s,F_s} \tag{2.10}$$

They waste some time in searching for the source what is described by the agent $Find_{s,F_s}$.[3] Afterwards the flight to the source is modelled as usual.

Comparison between real bees and the MAS Seeley mentions that the rate of scouting strongly depends on how many dancers are on the dance-floor. So does the MAS where bees which have failed to find a forager can become a scout. The aspect that foragers do not compare dances is modelled by $Follow \equiv \sum_{s=1}^{n_s} 1 : show_s.Find_{s,F_s}$; $\sum_{s=1}^{n_s}$ means a recruit chooses one of n_s food sources but *not* in a *deterministic* fashion. In the BeeAdHoc MAS this modelling is also used in the forager and packer agents.

[3] $Find_{s,F_s}$ is similar to Go_{s,G_s} with the parameters named differently

probabilities	
p_{r_s}	a forager dies or abandon her food source s
p_s	a forager of source s dances the waggle dance
p_j	additional delay of a flight
p_k	time wasted in searching for an advertised food source
p_o	a bee which does not find a forager on dance-floor wants to get another task at hive
p_f	bee in the hive visits the dance-floor
p_c	a recruit searching for the advertised food source finds it
p_t	a bee which does not find a forager on dance-floor becomes a scout
p_{l_s}	scout discovers source s

actions	
$\surd$	do nothing
$\overline{dance}$	perform the waggle dance
$dance$	follow a waggle dancer
$\overline{show_s}$	teach the location of source s
$show_s$	be taught the location of source s
$pick_s$	pick nectar from source s

Figure 2.1.: Variablen

2.3. Scouts

The model bases on the assumption that scouts are bees which have not found a forager on the dance-floor to be recruited. A scout can discover a food source s with a probability of p_{l_s}. l codes her perseverance in the search.

$$Scout \equiv \sum_{s=1}^{n_s} p_{l,s} : \surd.Evaluate_s + (1 - p_{l,1} - \ldots - p_{l,k_s}) \tag{2.11}$$

If source s is closer to hive $p_{l,s}$ should be greater. A search time is not included in the model.

$$Evaluate_s \equiv \omega : pick_s.Succeed_{s,R_s} + 1 : \surd.Scout \tag{2.12}$$

A scout tries to fetch food from the source she previously discovered. If she gets nectar she returns home as a forager (*Succeed*), if not she keeps scouting.

3. Implementation of BeeAdHoc in ns-2

3.1. Introduction

As the development and evaluation of routing protocols with real world MANETS is expensive and very difficult to do, we made use of network simulations. A key to achieve the desired results in reality, is to have a good simulation model of the targeted environment. Several network simulators are available, such as ns-2 [ns2] and Omnet++ [omn]. They aided us in evolving the Beehive routing algorithm.We use ns-2 for evaluation. It is a complete simulation environment with tested algorithms and methods.

3.1.1. Ns-2

Ns-2 is a discrete event simulator targeted at networking research. Ns-2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. [ns2] The following features of ns-2 helped us in utilizing it in our experiments:

- extensive TCP/IP stack (FullTCP, similar to a 4.x BSD stack)
- various traffic generators (CBR, VBR, FTP, HTTP, stochastic)
- Visualization of nodes and data [nam]
- Complete implementation of the IEEE 802.11 DCF MAC protocol
- Complete implementation of the Address Resolution Procotol (ARP)
- Implementation of DSR, DSDV, TORA and AODV
- Wireless network interface modeling the Lucent WaveLAN DSSS radio
- Modeling of signal attenuation, collision, and capture
- Simple energy model
- The scripting language OTcl to set-up the scenarios

3.2. OTcl

Ns-2 makes a twofold approach: the algorithm is written in C++ for speed and effciency, while the scenario configurations are described in OTcl [otc]. Furthermore the complete class hierarchy is available to both parts, which lets a user easily manipulate different aspects of C++ objects in OTcl, or extend the simulator with newly written scripts, with the penalty of execution speed and memory requirements. Using a programming language to control the simulation eases the set-up in complex cases. but a little effort is required to glue both worlds together. A short (non-working) TCL batch script to run a BeeAdHoc simulation may look like this:

```
set val(rp) BeeAdHoc
Agent/BeeAdHoc set VERBOSE 0
set ns_ [new Simulator]
set tracefd [open "val(rp).tr" w]
ns_ use-newtrace
ns_ node-config -adhocRouting val(rp)
ns_ run
```

After setting up variables (line 1), an option is set in the BeeAdHoc Agent. Line 3 instances the global Ns-2 object, which is the simulator itself. The next line opens a file descriptor (the last part of the command) and binds it to the variable tracefd. In line 4 ns is told to generate a trace file in the new trace format. . The next line is one command, which sets up the mobile node framework. Finally the event scheduler starts and the simulation is launched. The command ns sample.tcl results in a trace file called BeeAdHoc.tr, a NAM visualization le called BeeAdHoc.nam and a lot of debugging messages on stdout (and hopefully nothing on stderr). The trace file can be analyzed with parser.pl and BeeAdHoc.tr and the scenario can be executed with BeeAdHoc.nam.

3.3. Scenario generation

Although there are tools to generate movement and traffic patterns, we decided to use Ns-2's scripting abilities. The tools we found generated incorrect files or were just not usable. Since our goal was not to test out various movement patterns but to evaluate the routing algorithms, the simple but generally applicable random waypoint method was used. Our implementation is straightforward and written into the TCL scripts. We made use of the fact, that Ns-2 has a built-in PRNG (pseudo random number generator), which will generate the same sequence of numbers when fed with the same seed. The

PRNG is independent of the machine's hardware and the Operating System, so that a run with the same seed and options is exactly reproducible. Furthermore ns-2 is capable of instancing several independent PRNG streams, so that we do not have to take care if other parts of the simulator would disturb the contingency of the sequence. This sequence is used to initially position every node to a (pseudo) random (x, y) value. After that a function is called for every node, which does the following:

```
proc move_node {n} {
#set new x-value destination
set rnd_pos_x [expr [rnd value] * val(x)]
#set new y-value destination
set rnd_pos_y [expr [rnd value] * val(y)]
#the speed with which the node should move,
# in the range between rwpmin and rwpmax
set rnd_speed [expr [rnd value]]
set rnd_pos_s [(rnd_speed * val(rwpmin)) +
((1 - rnd_speed) * val(rwpmax))]
#execute the movement
ns_ at [ns_ now] "node_(n)␣setdest
#wait␣0␣to␣rwppause␣seconds␣and␣start␣movement␣again,
#␣eventually␣interrupting␣an␣ongoing␣movement
ns_␣at␣[expr␣[rnd␣value]␣*␣val(rwppause)+
[ns_now]]␣"move_node n"
}
```

The rnd variables are uniformly distributed floating point numbers between 0.0 and 1.0. To generate traffic we used the built-in CBR traffic agent over full TCP in most of our evaluations. FTP traffic generation and UDP were also tested. The nodes are connected in the following way: The rst node 0 sends its data to the last node n, the second node 1 to n-1 and so on, up to m nodes generating the traffic. In our results m was set equal to n. This scenario ensured that we did not generate any artificial hotspots, equally to a peer-to-peer network. That has the advantage of a comparable algorithms of every node and a standard deviation which is meaningful when composing the routing itself.

3.4. BeeAdHoc implementation

The main implementation of BeeAdHoc consists of the following files:

- adbeeDefinitions.h, defines constants and the structures of the agents. Most of the constants have been made available to the TCL hierarchy as variables for convenience.
- hdrbeeadhoc.cc hdrbeeadhoc.h links the BeeAdHoc packet header into the ns2 structures that is the source routing and IP options data felds.
- beeadhoc.cc beeadhoc.h the main class, responsible to handle most of the simulator specific details to enable easy porting of BeeAdHoc.
- beeadhoc.h beeadhoc's global definition file with data structures based on STL [stl] and class definitions.
- beeadhoc.tcl sets the TCL variables of BeeAdHoc to default values.
- adentrance.cc .h is the class, responsible for realizing entrance.
- adpackingfloor.cc .h the class, that is responsible for the realizing packingfloor.
- adbeefloor.cc .h the class, that is responsible for the dancefloor of BeeAdHoc.

To integrate BeeAdHoc into ns-2, some sources of the simulator had to be extended, with changes mostly one line.

- Makefile.in, of course our sources should be compiled too
- packet.cc, adds the BeeAdHoc header, defined in hdr BeeAdHoc to the core.
- priqueue.cc, to add the prefer routing protocols e.g., although unused in our simulations
- tcl/lib/ns-lib.tcl, this adds the BeeAdHoc routing agent to the TCL hierarchy.
- tcl/lib/ns-packet.tcl, which makes our packet header available.

The class BeeAdHoc.cc provides, as mentioned, an interface to the simulator. It instantiates upon creation the packetfloor, entrance and dancefloor, and binds the TCL variables. BeeAdHoc::command accepts the TCL events sent by the core, as setting the local address per node, or attaching objects like the port demux. but most of the events are handled by BeeAdHoc's parent class, the ns-2 agent. BeeAdHoc::recv takes a packet as an argument and forwards it, depending on the header, to handleFromApp, handleScout or handleForager. These and other functions in BeeAdHoc.cc are helper functions to transform ns-2 packets in BeeAdHoc agents and vice versa. Since these are simple pointer operations or API specific details (or because of the lack of documentation about ns-2's internals), there is no reason to explain this further. Finally the packets will traverse into the packingfloor, if they are to be sent; into the entrance, if they are coming from the net; or send/broadcasted out into it.

One detail of the simulation is noteworthy: a problem arose, when we broadcasted our scouts into the neighborhood, a fast queue built up was experienced. Since the transmission time and delays between the wireless nodes are exactly the same when no problem occurs, all neighbor nodes broadcasted their scouts again at the same time, which led to a massive collision of the scouts. This would not happen in reality, so we added a small random delayer upon receiving and forwarding a scout on every node. It appears that the DSR implementation does the same.

3.5. Implementation of Swarm cocepts in ns-2

Swarm are control packets. They are needed when a communication link isn't balanced in both direction in terms of the amount of data packets. This will lead to the sending node running out of foragers all the time.To avoid this the swarming technique is developed.

3.5.1. Swarms Model in BeeAdHoc

Swarms Model consists of the following Agents in BeeAdHoc:

- SwarmLeader
- SwarmRecruiter
- SwarmMember

The agent swarmleader decides the senderate. If the transmission rate of a node n increases, the senderate can be increased by the packer. If the transmission rate is too low, because too many foragers arrive, the senderate can be reduced. When the sendrate is too low, the swarmleader starts and many foragers are to be sent as SwarmMember.

SwarmRecruiter contains the identification of its source and destination node and search a forager, which can be sent as a SwarmMember in a swarm. They are to replace missing packer, however they carry no datapacket and cannot be recruited by dances.

SwarmMember is a set of foragers, that are sent in a swarmpacket.

If a SwarmRecruiter and its associated SwarmMember arrived at the destination node, the SwarmMember is unpacked in the destination node. if a SwarmRecruiter is lost on a route ,then its SwarmMembers are also deleted.

3.5.2. Swarms in ns-2

Swarmpakets are created in the BeeAdHoc model on dancefloor , therefore they are implemented in adbeefloor.cc .The main implementation of swarms consists of the following functions:

- addSwarmEntry()
- addSwarm()

The addSwarmEntry function implements SwarmLeader and SwarmRecruiter. The function is called periodically. It looks for a forager, which has a higher waiting time in the forgerlist and add this forager as SwarmMember in swarm. If sendrate is too low, the SwarmRecruiter is sent as forager with SwarmMemeber.

```
        AdForager* bee;
    for(list<AdSwarmEntry*>::iterator swarmBee =
            swarmLeader->swarm->bees.begin(); swarmBee!=
            swarmLeader->swarm->bees.end();swarmBee++)
    {
         bee = new AdForager();
      for (deque<int>::iterator i=(*swarmBee)->route.
           begin(); i!=(*swarmBee)->route.end(); i++)
           bee->route.push_back((*i));
      bee->kind = (*swarmBee)->kind;
      bee->info = (*swarmBee)->info;
      bee->ns2_p = NULL;
      bee->arrivaltime = BeeAdHoc->simTime();
      bee->justArrived =(*swarmBee)->justArrived;
      bee->dancetime = (*swarmBee)->dancetime;

      addForagerToDestListNode(bee, dln);

      delete (*swarmBee);
    }

    delete swarmLeader->swarm;
    swarmLeader->nrPacketsWaiting = (int)
```

```
        (swarmLeader->nrPacketsWaiting/swarmLeader->info);
    swarmLeader->nrDances = rateFlight(swarmLeader);
    swarmLeader->kind = ENERGY;
    addForagerToDestListNode(swarmLeader, dln);
```

addSwarm function unpacks the arrived swarm.The swarmmember is inserted as forager in the foragerlist.

```
      for(list<AdSwarmEntry*>::iterator swarmBee =
          swarmLeader.swarm.bees.begin();swarmBee!=
          swarmLeader.swarm.bees.end();  swarmBee++)
    {
      bee = new AdForager();
      for (deque<int>::iterator i=(*swarmBee).route.
               begin(); i!=(*swarmBee).route.end(); i++)
            bee.route.pushback((*i));
        bee.kind = (*swarmBee).kind;
        bee.info = (*swarmBee).info;
        bee.ns2p = NULL;
        bee.arrivaltime = BeeAdHoc.simTime();
        addForagerToDestListNode(bee, dln);

        delete (*swarmBee);
    }
```

4. Mobile Agent in BeeAdHoc

Our model is based on the principles of decentralization and locality. So our design concentrates on the local processes in one node and not covers the issues in a whole ad hoc network at all. [1] With the routing information, gained by the *scouting*, the BeeAdHoc Agents should be capable of routing with rather optimized lifetime, energy, throughput, delay, overhead and packet lost; but our focus is on lifetime and multiobjective optimization has not been dealt with yet. We intended to improve BeeAdHoc by making it more adaptable to a changing network, e.g. changes in routes or difference in sending rates in UDP where additional control packages are needed in BeeAdHoc to maintain routing information. A better adaptability should decrease the overhead.

4.1. Forager

A forager is the most important agent in the routing since it influences the choices of routes by her dancing behaviour if some routes already have been discovered by scouts. The routing principles have been adopted from the previous version of BeeAdHoc and has merely been formalized in the WSCCS model : the numbers of foragers using a certain route is proportional to the quality of the routes as evaluated by the foragers. A new dimension is added to the model because an adaptability to the number of foragers is needed. Foragers should dance with less vigour if the requirement for more foragers decreases. This aims to reduce the total numbers of foragers created in the network.[2] A forager is also involved in creating swarms of foragers which is described in the section *Swarms* in more detail.

A formal model There are different types of forager agents: Both $ForagerAtHive$ and $WaitingForager$ know the identity number of their source and destination nodes (n, d), one route between them under an unique ID $(i_{n,d})$ and its value RV. The other two forager agents $RecruitingForager$ and $DancingForager$ also have

[1] see paragraph *The network* in section *Forager*

[2] Bees encounter the same situation as well when there is a lack of food-storer bees. Then the bees also dance with less vigour in order to avoid more nectar being collected.

a remaining dance time dt and remember the waiting time for their first packet received.[3]

The following agent describes a forager, which is still waiting for her first packet:

$$ForagerAtHive(n, d, i_{n,d}, RV, T_{FW}) \equiv$$

$$\begin{cases} \omega^2 : \overline{searchForager_{(n,d)}}. \begin{cases} OnRouteForager(n, d, i_{n,d}) \\ \text{find } dt^* = 0 \\ RecruitingForager(n, d, i_{n,d}, RV, T_{FW} + 1, dt - 1) \\ \text{find } dt^* > 0 \end{cases} \\ \omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n, d, i_{n,d}, RV) \\ +1 : alert_{(n,d)}.SwarmMember(n, d, i_{n,d}, RV) \\ \text{for } T_{FW} = \text{F\_MAX\_WT} \\ \omega^2 : \overline{searchForager_{(n,d)}}. \begin{cases} OnRouteForager(n, d, i_{n,d}) \\ \text{find } dt^* = 0 \\ RecruitingForager(n, d, i_{n,d}, RV, T_{FW} + 1, dt - 1) \\ \text{for } dt^* > 0 \end{cases} \\ +\omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n, d, i_{n,d}, RV) \\ +1 : \surd.ForagerAtHive(n, d, i_{n,d}, RV, T_{FW}) \\ \text{otherwise} \end{cases} \tag{4.1}$$

She measures her waiting time T_{FW} until she finally receives a message ($\overline{searchForager}$), and uses this time and the value of her route RV to evaluate her dance time*:
If the waiting time crosses a global limit F_MAX_WT, she alerts the agent $SwarmSender$ and is sent in the next swarm packet ($alert_{(n,d)}.SwarmMember$).

A forager, which is recruiting a packer is described by the following agent:

[3] The WSSCS model does not care about the forager numbers being limited like in NS2 (a confined forager list). For different reasons forager numbers had to be limited in the mean difference equations simulation. Otherwise the number of foragers grew so high that the simulation could not deal with it any longer and swarms were sent on TCP. This should be avoided.

*
$$dt = max(RV * (1 - (\frac{T_{FW} + 1}{\frac{\text{F\_MAX\_WT}}{2}})^2), 0)$$

$$\begin{aligned} &RecruitingForager(n, d, i_{n,d}, RV, T_{FW}, dt) \equiv \\ &1 : \overline{gainRouteInfo_{i_{n,d}}}.DancingForager(n, i, RV, T_{FW}, dt) \end{aligned} \tag{4.2}$$

A dancing forager tries to recruit a packer :

$$DancingForager(n, d, i_{n,d}, RV, T_{FW}, dt) \equiv$$

$$\begin{cases} \omega^2 : \overline{searchForager_{(n,d)}}.OnRouteForager(n, d, i_{n,d}) \\ +\omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n, d, i_{n,d}, RV) \\ +1 : \surd.WaitingForager(n, d, i_{n,d}, RV, T_{FW} - 1) \\ \text{for } dt = 0 \\ \omega^2 : \overline{searchForager_{(n,d)}}.RecruitingForager(n, d, i_{n,d}, RV, T_{FW}, dt - 1) \\ +\omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n, d, i_{n,d}, RV) \\ +1 : \surd.DancingForager(n, d, i_{n,d}, RV, T_{FW}, dt - 1) \\ \text{find } dt > 0 \end{cases} \tag{4.3}$$

In the recruitment the two communicate by $\overline{searchForager}$.
Foragers, which have not been found by a packer, still can become member of a swarm ($\overline{recruitSwarm}$).

If the dance time has expired, a forager keeps waiting for a packer or becoming a swarm member:

This equation is close to the behaviour of the foragers observed during their dances in nature: there the dance time depends on their search time for a food-storer bee. (see figure 1.1 on page 4)

$$WaitingForager(n,d,i_{n,d},RV,T_{FW}) \equiv$$

$$\begin{cases} \omega^2 : \overline{searchForager_{(n,d)}}.OnRouteForager(n,d,i_{n,d}) \\ +\omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n,d,i_{n,d},RV) \\ +1 : \surd.Deleted \\ \text{for } T_{FW}=0 \\ \omega^2 : \overline{searchForager_{(n,d)}}.OnRouteForager(n,d,i_{n,d}) \\ +\omega : \overline{recruitSwarm_{(n,d)}}.SwarmMember(n,d,i_{n,d},RV) \\ +1 : \surd.WaitingForager(n,d,i_{n,d},RV,T_{FW}-1) \\ \text{for } T_{FW}>0 \end{cases} \tag{4.4}$$

She waits as long as she did for her first packet before she retires from work. (see $ForagerAtHive$).

A forager may ask the value of her route on arrival at the destination by $getValue_{i_{n,d},RV}$ if a value of a route is changeable during a scenario. If there is a fixed route value $\sum^{max_{RV}} getValue_{i_{n,d},RV}$ can be replaced by one $arrive_{i_{n,d}}$:

$$\begin{aligned} &OnRouteForager(n,d,i_{n,d}) \equiv \\ &\omega : \textstyle\sum^{max_{RV}} getValue_{i_{n,d},RV}.ForagerAtHive(d,n,i_{n,d},RV,0) \\ &+1 : \surd.Deleted \end{aligned} \tag{4.5}$$

$\sum^{max_{RV}}$ is necessary in order to catch any possible route value.

A deleted agent:

$$Deleted \equiv 1 : \surd.Deleted \tag{4.6}$$

The network There is no network topology described within the model yet; there are fixed routes with fixed values which can be discovered by a scout with a fixed probability. The traffic on routes is *not* supposed to influence one another.
The route agent plays a similar role like the food agent in equation 2.4 on page 12, especially a capacity is needed in order to confine the number of foragers :

$$Route(i_{n,d}, RV) \equiv \sum_{j=1}^{c_r} \omega : \overline{arrive_{i_{n,d}}}^{j} . \overline{getRouteValue_{i_{n,d}}(RV)}^{j} . Route(i_{n,d}, RV) \tag{4.7}$$

4.2. Packer and Swarms

Our model closely follows its counterpart in Nature. A bee colony can adapt its nectar collecting rate and nectar processing rate :
On the one hand the number of foragers rises, if there is a source with plenty of food. On the other hand, if the nectar unloading has slowed down, the number of food-storer bees is increased. Which type is actually recruited is decided by the foragers according to the *search time* for a food storer bee they have experienced.
Like in the previous version of*BeeAdHoc* we regard a node as a hive and a route as a food source. In our agent model a *forager* decides according to her *waiting time* for a packer[4], whether she recruits new foragers by dancing or if the incoming rate of foragers is greater than the rate of incoming packers. In this case *swarms* try to maintain a balance between these two bees.

4.2.1. Packer

A packer receives a packet from the *transport layer*. The transport layer is quite simple: an agent $PackerSender(n, d, pR_{(n,d)})$ creates $pR_{(n,d)}$ packers in one time step. The *packers* search (in the list of foragers of their source node n) for a forager flying to the required destination d, as long as their lifetimes have not elapsed.($searchForager_{(n,d)}$) Otherwise they are deleted.
Among all foragers one is chosen by a packer randomly with equal distribution ($gainRouteInfoi_{(n,d)}$) and the packer is recruited, if the chosen forager is still dancing.[5]
If a packer cannot find any forager for her desired destination, she tries to reduce the sending rate of swarms ($stopSwarms_{(n,d)}$). But in case this rate is already zero, a scout immediately is broadcasted.

A formal model A packer knows the ID of both her source and destination node and carries a data packet given to it by the transport layer.

[4] Like in the previous version of BeeAdHoc packers are agents, which receive one packet from the upper layer and try to deliver it to a forager, which delivers the packet to its destination.
[5] In the BeeAdHoc algorithm the packer is deleted and an appropriate forager created

n	ID of the source node
d	ID of the destination node
$i_{n,d}$	ID of the route between n and d which the forager uses
max_{RV}	maximum value of a route
Packer	
P_MAX_WT	waiting time before a packer is deleted
$pR_{(n,d)}$	number of packets addressed to node d and received from the upper layer at node n in one time step
swarms	
RV	value of the route the forager is using
F_MAX_WT	the maximal waiting time of a forager
T_{FW}	current waiting time of the forager
T_D	remaining dance time
sRn, d	foragers which are sent from n to d in one time step
$minS$	number of foragers which at least need to be in a swarm packet
$maxS$	maximum number of foragers which can be included in a swarm packet
scouts	
Sc_{TTL}	initial lifetime of a scout

Figure 4.1.: Variables

Data packets are not explicitly coded in the model:

$$PackerSender(n,d,pR_{(n,d)}) \equiv \begin{cases} 1 : \surd.PackerSender(n,d,0) \\ \text{for } pR_{(n,d)} = 0 \\ 1 : \surd.\prod_{i=1}^{pR_{(n,d)}} Packer(n,d,0) \times PackerSender(n,d,pR_{(n,d)}) \\ \text{otherwise} \end{cases} \tag{4.8}$$

The behaviour of a packer is described by the following agent:

$$Packer(n,d,wt) \equiv \begin{cases} \omega^2 : searchForager_{(n,d)}.Recruit(n,d) \\ +\omega : stopSwarms_{(n,d)}.Deleted \\ +1 : \surd.Deleted \\ \text{find } wt = \text{P\_MAX\_WT} \\ \omega^3 : searchForager_{(n,d)}.Recruit(n,d) \\ +\omega^2 : stopSwarms_{(n,d)}.Packer(n,d,wt+1) \\ +\omega : \surd.Packer(n,d,wt+1) \times Scout(n,d,Sc_{TTL}) \\ +1 : \surd.Packer(n,d,wt+1) \\ \text{otherwise} \end{cases} \tag{4.9}$$

$$\begin{aligned} &Recruit(n,d) \equiv \\ &\textstyle\sum_{i_{n,d}} \omega : gainRouteInfo_{i_{n,d}}.OnRouteForager(n,d,i_{n,d}) \\ &+1 : \surd.Deleted^{6} \end{aligned} \tag{4.10}$$

A packer is interested in delivering her data packet in the first place (*searchForager*). To send as less swarms as possible if packers are available, a packer *always* informs the *SwarmSender* whether she found no forager (*stopSwarms*).

4.2.2. Swarms

The agent *SwarmSender* controls, how many foragers are sent as swarm members in one time step. If the sending rate of packets of node n increases again, the rate of swarm members to be sent can be lowered (j times) by packers ($\overline{stopSwarms}^{j}$). If the incoming foragers outnumber the data packets to be sent, the rate of swarm members to be sent

[6] Usually a packer becomes a forager and gains the routing information from her recruiter which had advertised the route to the packer. But if this recruiter instead of recruiting delivers the packet on her own, the packer is deleted.

is increased; in this situation foragers are alerting the swarm sender that the number of packets is too small ($alert_{n,d}$), if their maximum waiting time has expired without having seen any packer (by $\overline{searchForager}_{n,d}$).
A *SwarmRecruiter* is looking for a forager to recruit her to a *SwarmMember* which can be sent within a swarm packet. Swarm recruiters should replace missing packers, but they do not carry any data packet and also cannot be recruited by dances. Like packers they have a lifetime as well, which controls their life time. And like packers they randomly choose among the foragers with equal distribution($recruitSwarm_{(n,d)}$), but are treated with a lower priority than the data packets. A *SwarmMember* is a forager which has to be included in a swarm packet. Swarm members should never be resent in a swarm so that they do not keep on flying between two nodes.

A formal model $SwarmRecruiter$ have the ID of their source node and their desired destination. $SwarmMember$ agents are similar to *foragers* and can be grouped in a swarm in a limited number; a *swarm* is like a $OnRouteForager$, which carries other foragers as swarm members in her data packet.
A $SwarmSender$ resembles the $PackerSender$, though its sending rate can be changed:

$$SwarmSender(n,d,sRn,d) \equiv \begin{cases} \sum_j \omega : \overline{alert_{(n,d)}}^j.SwarmSender(n,d,sR_{(n,d)}+j) \\ \text{for } sR_{(n,d)} = 0 \\ \sum_k \omega^2 : \overline{stopSwarms_{(n,d)}}^k.SwarmSender(n,d,max(0,sR_{(n,d)}-k)) \\ +\omega : \surd \cdot \prod_{l=1}^{sR_{(n,d)}} SwarmRecruiter(n,d,0) \\ +\sum_j 1 : \overline{alert_{(n,d)}}^j.SwarmSender(n,d,sR_{(n,d)}+j) \\ \times SwarmSender(n,d,sR_{(n,d)}) \\ \text{otherwise} \end{cases} \tag{4.11}$$

A $SwarmRecruiter$ is described by the following agent:

$$SwarmRecruiter(n,d,wt) \equiv \begin{cases} \omega : recruitSwarm_{(n,d)}.Deleted \\ +1 : \surd.Deleted \\ \text{find } wt = \text{S\_MAX\_WT} \\ \omega : recruitSwarm_{(n,d)}.Deleted \\ +1 : \surd.Deleted \\ \text{otherwise} \end{cases} \tag{4.12}$$

Packers are treated with a higher priority which is controlled by the forager agents. The following agent handles, how a swarm is to be sent:

$$\begin{aligned}
&SwarmMember(n,d,i_{n,d},RV) \equiv \\
&\omega : \overline{buildSwarm_{(n,d)}}.OnRouteSwarmMember(n,d,i_{n,d},RV) \\
&+\textstyle\sum_{m=minS}^{maxS} buildSwarm^{m}_{(n,d)}.SwarmLeader(n,d,i_{n,d},RV,m) \\
&+1 : \surd.SwarmMember(n,d,i_{n,d},RV)
\end{aligned} \tag{4.13}$$

The priorities guarantee that the swarm packet is built with the biggest size possible.

$$\begin{aligned}
&OnRouteSwarmMember(n,d,i_{n,d},RV) \equiv \\
&\omega : \surd.\overline{arrived_{(n,d)}}.SwarmMemberAtHive(n,d,i_{n,d},RV,0) \\
&1 : \surd.\surd.Deleted
\end{aligned} \tag{4.14}$$

$$\begin{aligned}
&SwarmLeader(n,d,i_{n,d},RV,m) \equiv \\
&\omega : arrive_{i_{n,d}}.arrived^{m}_{(n,d)}.SwarmMemberAtHive(n,d,i_{n,d},RV,0) \\
&1 : \surd.Deleted
\end{aligned} \tag{4.15}$$

A *SwarmLeader* and its packet (a bunch of *OnRouteSwarmMember*) should arrive at the same time at the destination node.[7] In case a swarm leader is lost on the route (*Deleted*), because the capacity of the route is exhausted, all the swarm member which accompany her are deleted, too. Otherwise a swarm is unzipped during $arrived^{m}_{(n,d)}$.

The following agent is needed to stop swarm members from being sent around the place:

[7] For that purpose the additional $\surd$ is used in $\surd.\overline{arrived}$. Writing *action.action* is an abbreviation, which meanings can be found in the appendix under WSCCS.

$$SwarmMemberAtHive(n, d, i_{n,d}, RV, T_{FW}) \equiv$$

$$\begin{cases} \omega : \overline{searchForager_{(n,d)}} . \begin{cases} OnRouteForager(n, d, i_{n,d}) \\ \text{find } dt^1 = 0 \\ RecruitingForager(n, d, i_{n,d}, RV, T_{FW} + 1, dt - 1) \\ \text{find } dt^1 > 0 \end{cases} \\ +1 : \surd . Deleted \\ \text{find } T_{FW} = \text{F\_MAX\_WT} \\ \omega : \overline{searchForager_{(n,d)}} . \begin{cases} OnRouteForager(n, d, i_{n,d}) \\ \text{find } dt^1 = 0 \\ RecruitingForager(n, d, i_{n,d}, RV, T_{FW} + 1, dt - 1) \\ \text{find } dt^1 > 0 \end{cases} \\ +1 : \surd . SwarmMemberAtHive(n, d, i_{n,d}, RV, T_{FW}) \\ \text{otherwise} \end{cases} \tag{4.16}$$

4.3. Scouts

Scouts have the job to discover routes to a specified destination. They are ordered by packers which have failed to find a forager and the *SwarmSender* does not send any swarms. Once a scout returns to her source node she becomes a forager.

A formal model A sent scout knows the IDs of her source and destination node and her lifetime (TTL). She additionally has information about the discovered route ($i_{n,d}$) and its value (RV) after arriving at the destination node.

A simple agent can describe a scout in the following way:

$$Scout(n,d,TTL) \equiv \begin{cases} 1 : \surd.Deleted \\ \text{find } TTL = 0 \\ \sum_{i_{n,d}} \sum^{max_{RV}} p_{i_{n,d}} : arrive_{i_{n,d}}.getValue_{i_{n,d}}(RV). \\ .ForagerAtHive(n,d,i_{n,d},RV,0) \\ +(1-\sum_{i_{n,d}} p_{i_{n,d}}) : \surd.Scout(n,d,TTL-1) \\ \text{otherwise} \end{cases} \tag{4.17}$$

The model ignores how a search for a route takes place, but a scout discovers one possible route from n to d with a probability of $p_{i_{n,d}}$ by calling $arrive_{i_{n,d}}$. The scout immediately returns to her source node without any delay ($ForagerAtHive(n,\ldots)$).

4.4. Checking the model by Simulation

The model developed has been supported by MDE simulations.[8] As MDE is rather a heuristic method, mistakes in understanding the model may occur. Therefore the model should also be checked by the Probability Workbench, which is designed for the analysis of agent written in our modelling language WSCCS. Its usage is described in the appendix. The Workbench builds a transition graph sticking to the rules of the WSCCS semantic, so a simulation run based on this graph is one possible behaviour of the model. The purpose of the simulation is either to reveal mistakes in the design or to support the features of the model, which are mentioned in the simulations below. The simulation also gives an insight into the behaviour of the model and therefore makes it more understandable.

4.4.1. Forager

The most important feature is the selection of routes according to the route evaluation of the foragers. The adaptability to mobility has been tested with the NS2 simulator and has not been analysed with the Workbench.

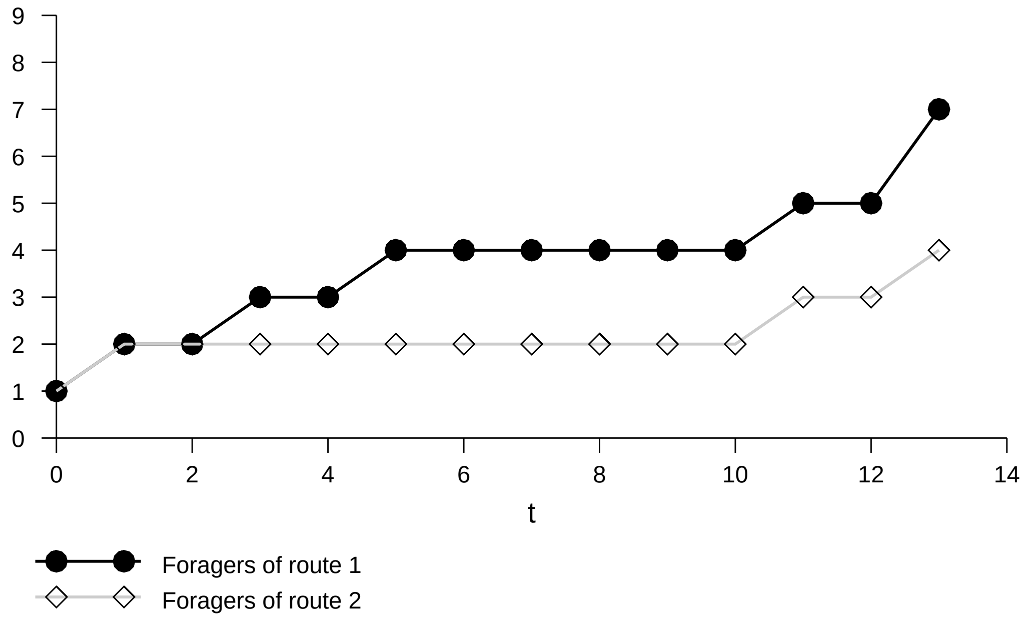
In the following scenario one UDP connection is established with two routes available. Route 1 has value 3 and Route 2 value 1. Because both routes have the same delay, not the delay but the battery is evaluated in this scenario. The scenario is

[8] See the *Zwischenbericht* for more details. The simulation is written in *Java* and can be found in the BeeAdHoc repository under BMAS/MDE including all results.

described by the following agent:

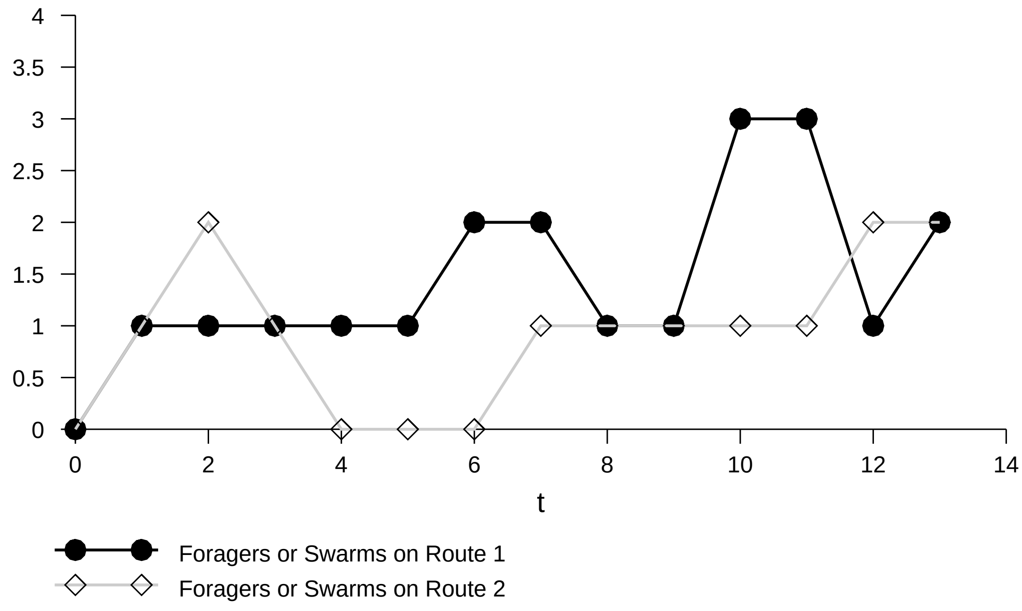
$$ForagerAtHive\_1\_2\_1\_3\_0|ForagerAtHive\_1\_2\_2\_1\_0|PackerSender\_1\_2\_2|$$
$$SwarmSender\_1\_2\_0|Route\_1\_2\_1|Route\_1\_2\_2|SwarmSender\_2\_1\_0$$

The simulation of the agent results in the following distribution of foragers on the two routes:[9]



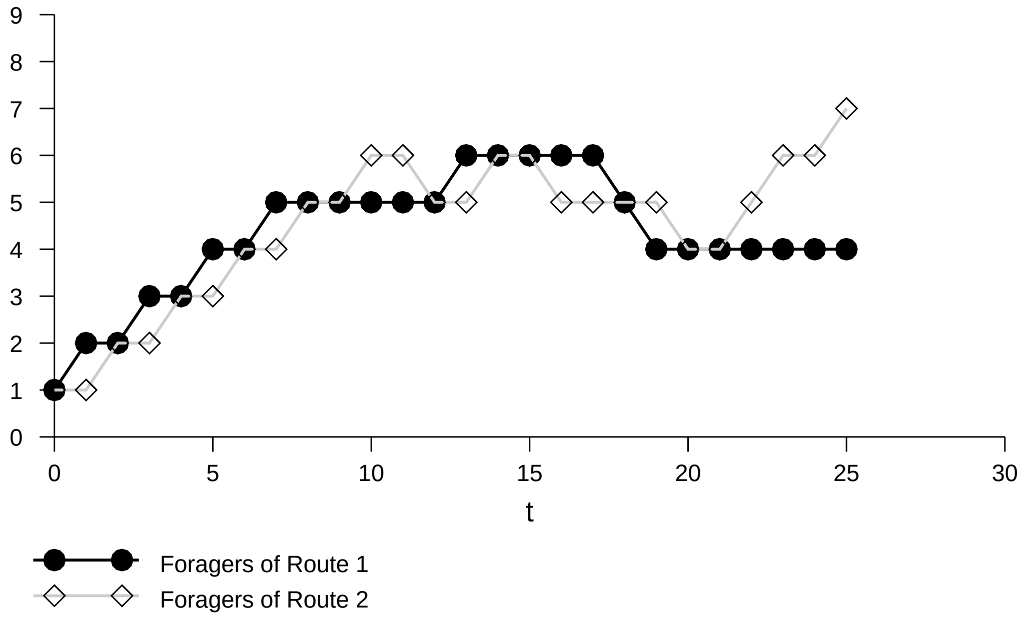
The figure indicates that the foragers may be distributed on the routes according to the rating of 3:1 for route 1. Nevertheless the traffic is not yet distributed this way, as the following picture shows:

[9] The simulation can be found in ./BMAS/Prob/BeeAdHoc/szenario13.prob and its result in ./B-MAS/Prob/BeeAdHoc/results/results13_3.txt on the PG CD.



As a conclusion of this simulation one can say there is a tendency that the model behaves in the expected way but the simulation time is too short to say more.

A further simulation supports the assumption that a route value cannot be used as a dance time without any adaptability to the altering range of route values like in the model or in NS2 as well. In both cases the dance time is estimated using the *absolute* route value.
The scenario is similar to the previously tested one, except the values of the routes were set to 10 for route 1 and 5 for route 2 and the send rate has been risen from 1 to 3 packets each time step after 20 simulation steps to increase the activity of sending. The agent TEST4 describing this scenario is too big to be presented at this place and can be looked up in the file ./BMAS/Prob/BeeAdHoc/szenario8.prob. The following figure shows the distribution of foragers on the two routes:



One can notice that the model behaves the opposite way than expected. Route 1 has been rated by 2:1 against route 2. But route 2 seems to be preferred, although the simulation ran nearly twice as long than in the previous scenario. As the values of both routes are quite high packers are recruited vividly to both routes and which one is preferred seems to be random.

In nature bees can change the range of the evaluation of food sources by a threshold which is adapted to the availability of nectar in the colonies surroundings. In a network one can determine this threshold by collecting static of the route values of incoming foragers and find out a suitable range of the dance time which should be fixed for the BeeAdHoc algorithm. Below the threshold there should be no dancing and above the threshold the route value has to be treated considering a maximum value which has to be derived from the static gained from the network.

4.4.2. Swarms

There are several features to be checked:

* the optimal waiting time is independent from other unfixed parameters like the sending rate of packets
* by swarming the sending rate of foragers is adapted to the following situations:

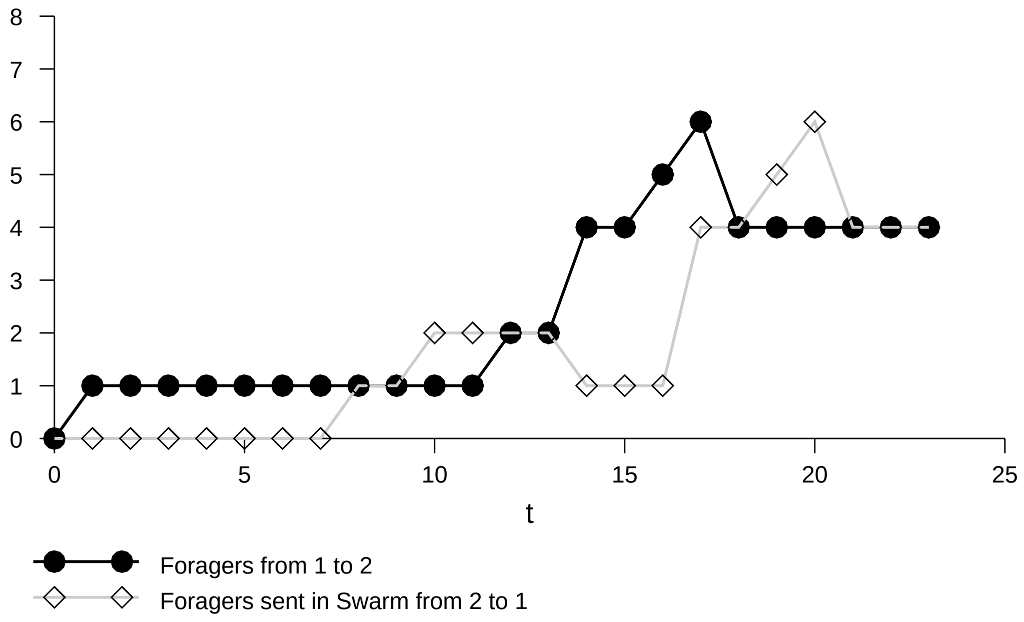
1. the destination node starts to send packets with a lower rate than the source
2. the sending of data packets increases during swarming
3. the source lowers its rate of data packets during swarming

Adaptability

1. An initial scenario has to be coded in a WSCCS agent (in the input format of the workbench), so the testing scenario is:

$$ForagerAtHive\_1\_2\_1\_10\_0|PackerSender\_1\_2\_2|SwarmSender\_1\_2\_0|$$
$$Route\_1\_2\_1|SwarmSender\_2\_1\_0$$

For the notation and naming of the agents see the appendix. In words the scenario is a one-way connection with two data packets being sent on each time steps and an initial forager is available in the list of the source node.[10]
The result of simulating the agent is presented in the following picture:



[10] The parameters of the scenario and their values can be found in ./BMAS/Prob/BeeAdHoc/szenario9.prob on the PG CD and the simulations results in ./BMAS/Prob/BeeAdHoc/results/results9_1.prob .

The numbers of foragers sent from 2 to 1 *not* in a swarm are zero during the whole simulation. The numbers of foragers which are resent at node 2 in a swarm get on the same level with the foragers sent to node 2. So the swarming seems to work well in this scenario. The resending of swarms should be *F_MAX_WT + route delay* time steps after starting the connection which is 4 time steps in this scenario. The additional delay results from the time the *SwarmSender* agent needs to respond. Since there are no packers which can interrupt the sending of swarms to lower it this response is nearly immediate.

2. The scenario is initialised with the final scenario where the simulation of 1. ended with. The simulation has not been completed so there are no results.[11] So the question still remains how the swarming and its interruption by packers work together.

3. Like in 2. the scenario is initialised with the final scenario where the simulation of 1. ended with and its simulation has not been done yet.[12]

Further Tests As mentioned above the optimal waiting time should be a constant and not dependent on any special scenario. So the simulation of paragraph 1. could be repeated with different packet send rates. To check if the *SwarmRecruiter* agent is necessary they can be replaced by a *SwarmSender* recruiting up to as many foragers each time step as the swarm send rate allows. *SwarmRecruiters* are supposed to keep the outgoing swarms on a rather constant rate. If there is any difference which really gives an advantage is another question since implementing the *SwarmRecruiter* agents leads to additional processing costs.

[11] The scenario is described in the agent TEST2 in the file ./BMAS/Prob/BeeAdHoc/szenario9.prob .
[12] The scenario is described in the agent TEST3 in the file ./BMAS/Prob/BeeAdHoc/szenario9.prob .

5. Simulation Results with Transport Protocol TCP

5.1. Changes in NS2

We had to adapt the BeeAdHoc algorithm to our formal model. Forager behaviour remained unchanged, only the estimation of the dancetime was improved.

BeeAdHoc has an identical dancetime for each forager and one estimates the number of bees which can be recruited by this forager. It means that in a time period represented by `FORAGER_DANCETIME` one can send the estimated number of foragers represented by `nrDances` . After `FORAGER_DANCETIME` the recruiting forager flies if she got a packet. After `FORAGER_LIFETIME` the forager will be deleted.

BeeAdHocBMAS estimates only the time to dance (`dancetime`) and the number of dances is determined by this time. The time to dance depends on collected information, the route size and the waitingtime for a new packet. The forager doesn't dance if she gets her first packet after `FORAGER_LIFETIME/2` and she will be deleted after `FORAGER_LIFETIME` (`FORAGER_LIFETIME` is set equal to the value of `F_MAX_WT` parameter of the model). So we replaced the method `rateFlight` with `estimateDancetime` and we had to change the moment at which the dance time is estimated. We couldn't do it immediately after the arrival of a forager because we needed the waitingtime for the first packet that's why we estimated it in the method `removeForager`.

5.2. Simulation Runs

We tested BeeAdHoc, BeeAdHocBMAS, DSR and AODV. The discription of DSR and AODV is given in one of the later chapters. For the tests we used the testing environment of the previous project group.

Run-Nr.	maxSpeed	pauseTime	packets/sec	packet size	nodes
1	5	60	10	512	50
2	10	60	10	512	50
3	15	60	10	512	50
4	20	60	10	512	50
5	15	60	30	512	50
6	15	60	60	512	50
7	15	60	100	512	50
8	15	30	100	512	50
9	15	1	10	512	50
10	15	60	10	1024	50
11	15	60	10	2048	50
12	15	60	10	4096	50
13	15	1	100	512	25
14	15	1	100	512	75
27	15	60	100	512	100
28	15	30	100	512	100
29	15	1	10	512	100

Table 5.1.: TCP-Runs for BeeAdHocBMAS (x=100), BeeAdHoc (x=0), DSR (x=300), AODV (x=600)

5.3. Testing Mobility Behaviour

5.3.1. Node Velocity

At first, the influence of different node movement behaviours was tested using four cases with maximum speeds v_{max} of 5, 10, 15 and 20 m/s and a minimum speed v_{min} of 1 m/s. Figure 5.1 describes the influence of movement speed on the energy that has been used to deliver one kB of user data to the desired destination in average. This includes as well the energy of the involved control packets as the energy consumption to send and receive this data. As one can see BeeAdHocBMAS consumes often less energy than BeeAdHoc. The reason can be partially the average hopcount for the way from the source to the destination. Figure 5.2 shows that the routes of BeeAdHocBMAS are a little bit shorter than the routes of BeeAdHoc.

Figure 5.3 describes the influence of movement speed on the average delay it takes to deliver a data packet from its source to its destination. Although BeeAdHocBMAS is a little bit slower than BeeAdHoc, it is much better than DSR or AODV.

Figure 5.4 describes the throughput achieved. The throughput of BeeAdHocBMAS is better than the throughput of BeeAdHoc if the maximum node's speed is not faster than 15 m/s.

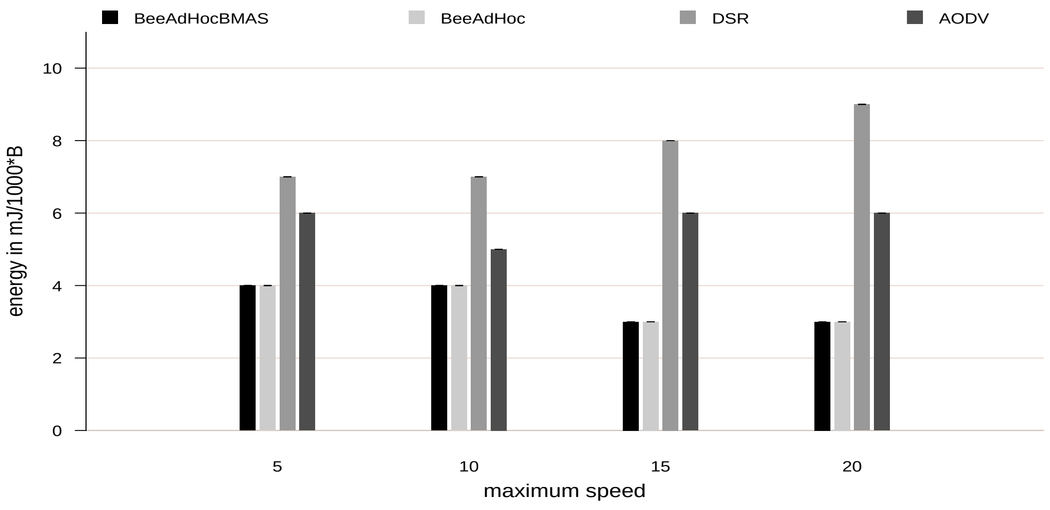


Figure 5.1.: Energy results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x)

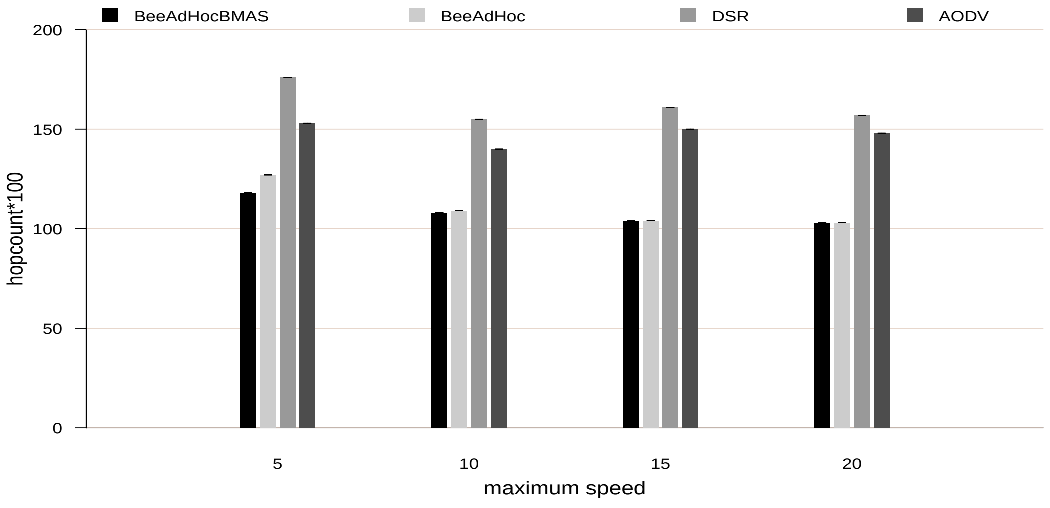


Figure 5.2.: Hopcount results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x)

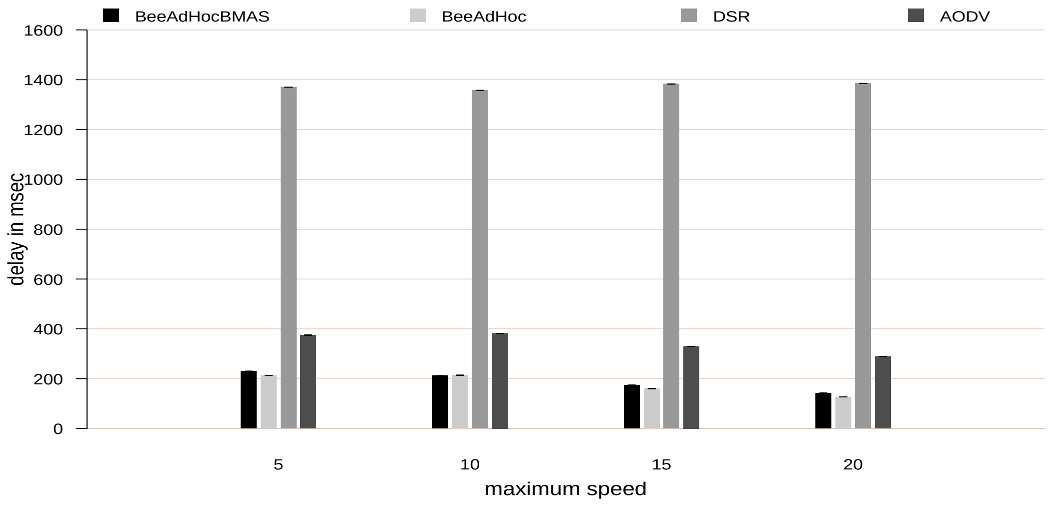


Figure 5.3.: Delay results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x)

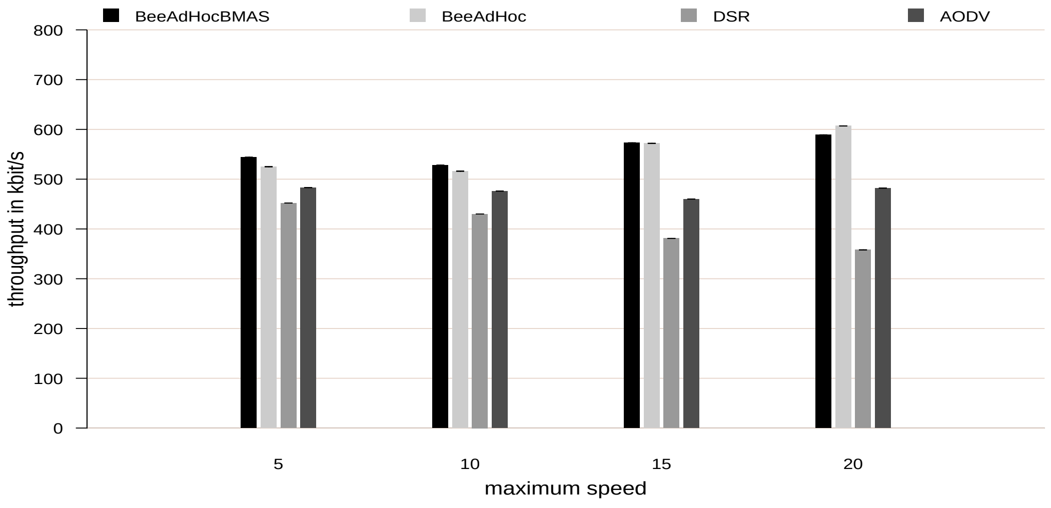


Figure 5.4.: Throughput results depending on node velocity(from runs 1+x, 2+x, 3+x, 4+x)

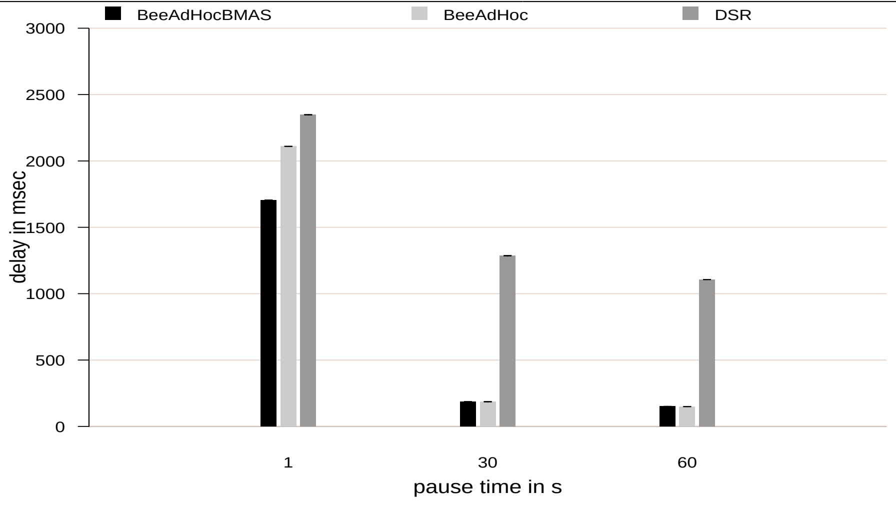


Figure 5.5.: Delay results depending on pause time (from runs 27+x, 28+x, 29+x)

5.3.2. Pause Time

Each node moves randomly to a specific destination and waits there for a time which is called pause time. Then it moves to another point, waits again and so on.

In the case of all the algorithms the delay is decreasing (see Figure 5.5). If a node rests for a longer time all the algorithms have got more time to adapt to this new situation. Routes exist longer, that's why we have higher throughput with increasing pause time (see Figure 5.6). Energy consumption decreases with increasing pause time because we need less route discovery (see Figure 5.7). DSR consumed so much energy that it is not comparable with the other algorithms and cannot be visible within the scale of this figure.

If we compare BeeAdHoc and BeeAdHocBMAS we can see that BeeAdHocBMAS is much better than BeeAdHoc if the pause time is shorter.

5.4. Testing Send Rate Behaviour

Sending more packets per time unit will result in a higher amount of packets sent and so in a higher amount of packets delivered successfully, which results in a higher throughput. This assumption is confirmed in Figure 5.8. Figure 5.9 shows that the energy consumption of each protocol is slightly increasing. The delay decreases a little bit (Figure 5.10). This is explainable by the fact that the initial route discovery time can be portioned to more overall packets. This fact also explains the decreasing number of

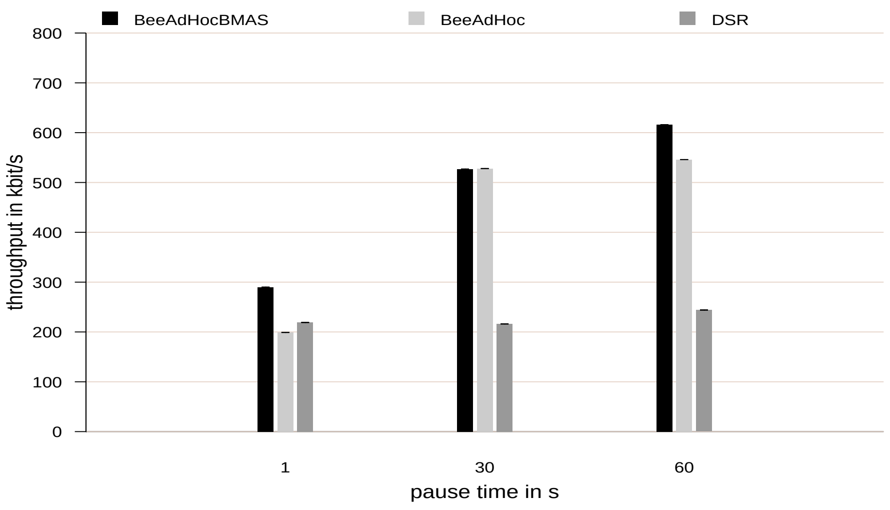


Figure 5.6.: Throughput results depending on pause time (from runs 27+x, 28+x, 29+x)

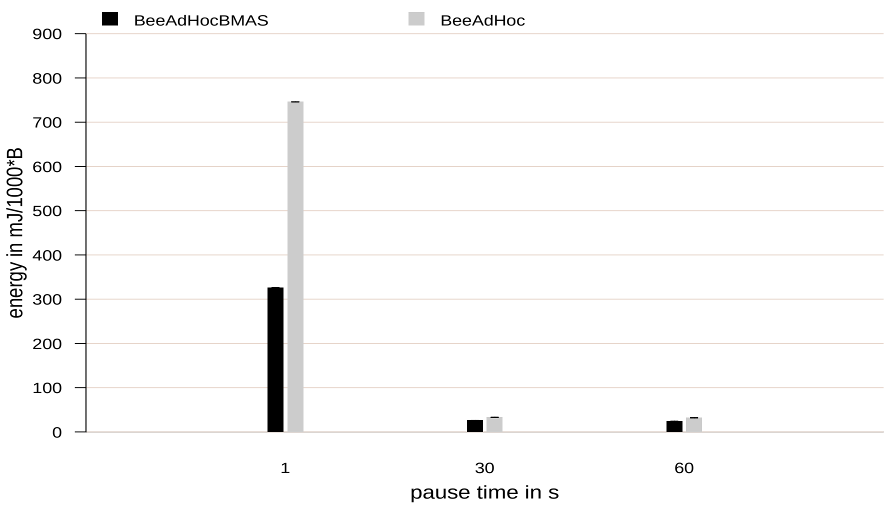


Figure 5.7.: Energy results depending on pause time (from runs 27+x, 28+x, 29+x)

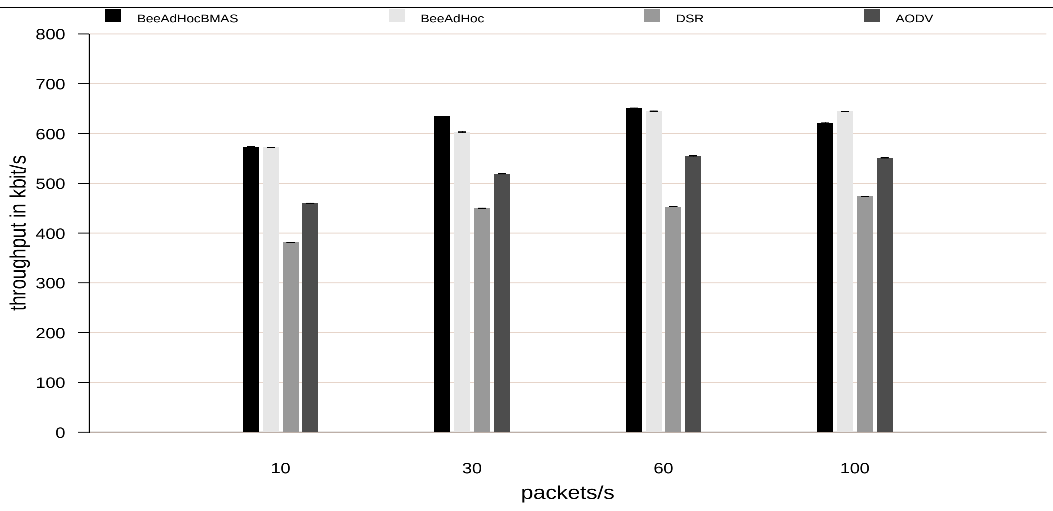


Figure 5.8.: Throughput results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x)

control packets by DSR and AODV. BeeAdHoc and BeeAdHocBMAS have the worst values in the runs with 30 and 100 packets/s (Figure 5.11).

5.5. Testing Packet Size Behaviour

Increasing the packet size means that the same amount of control packets can be portioned to more bytes of user data. That's why one needs less energy for sending control packets while one needs more energy to transmit one data packet. Figure 5.12 shows that in the cases of BeeAdHoc, BeeAdHocBMAS and AODV the energy consumption keeps nearly on a constant level. Only by DSR the energy consumption increases with increasing packet size.

The time needed to deliver one packet decreases with increasing packet size, because the time for route discovery can be portioned to more bytes of user data. This result is shown in figure 5.13 .

We expected increasing throughput by decreasing delay but one can see in figure 5.14 the throughput of all four protocols slightly decreased. This can be explained by the fact that the amount of successfully delivered packets also decreased. It means one delivered faster but not the same amount.

The amount of control packets does not depend on packet size (Figure 5.15)

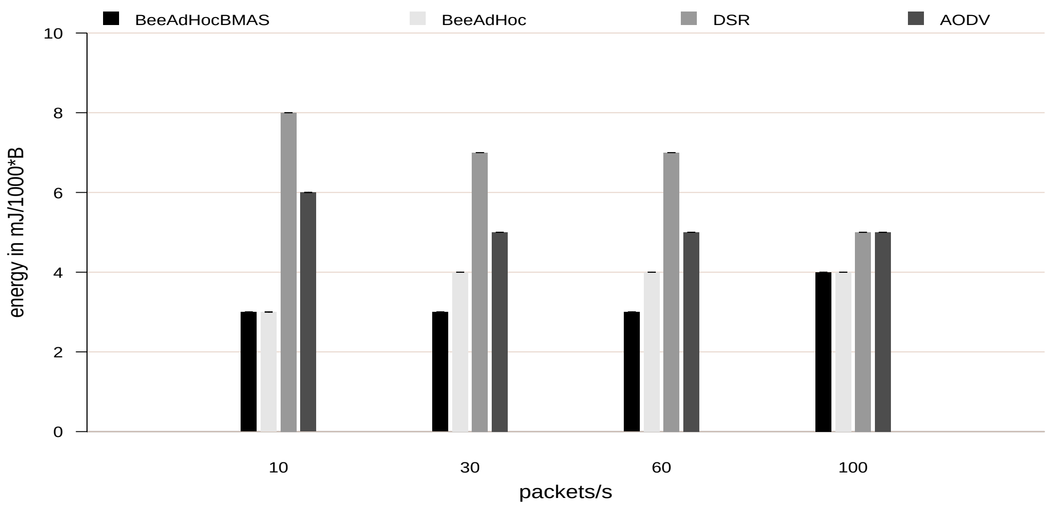


Figure 5.9.: Energy results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x)

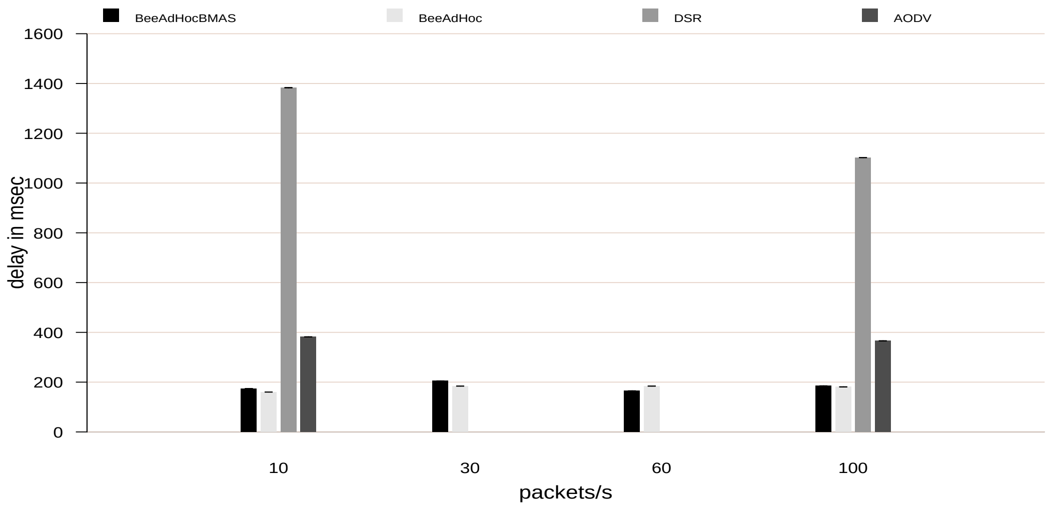


Figure 5.10.: Delay results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x)

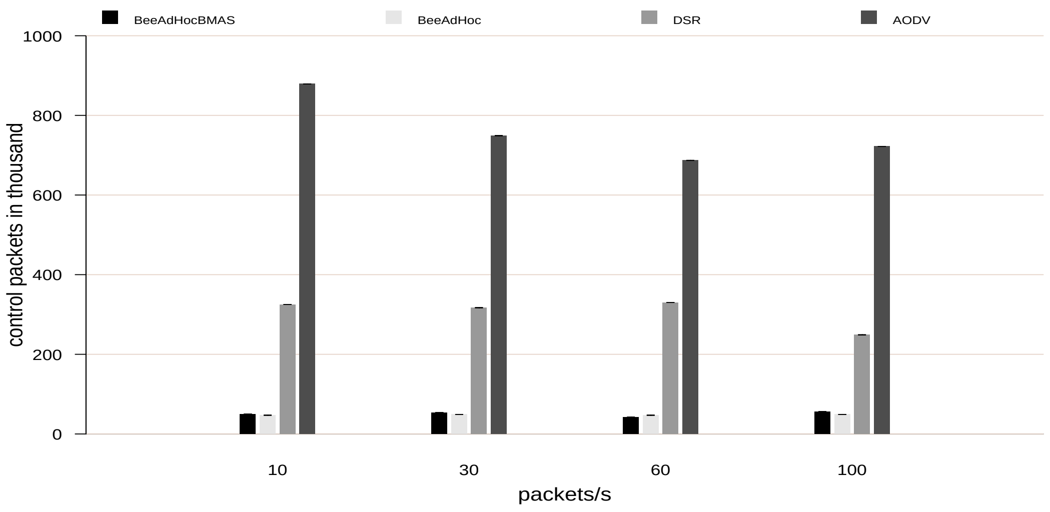


Figure 5.11.: Control packets results depending on the send rate (from runs 3+x, 5+x, 6+x, 7+x)

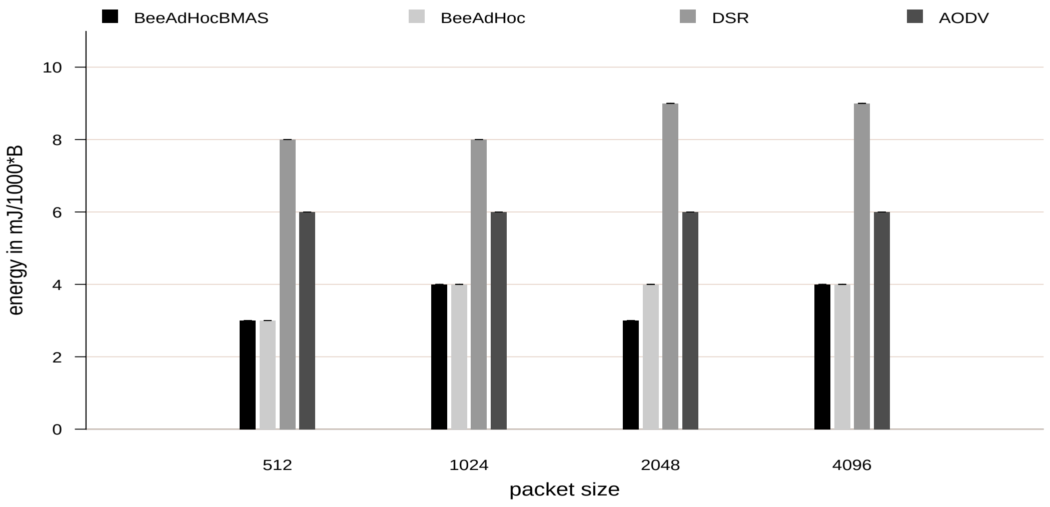


Figure 5.12.: Energy results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x)

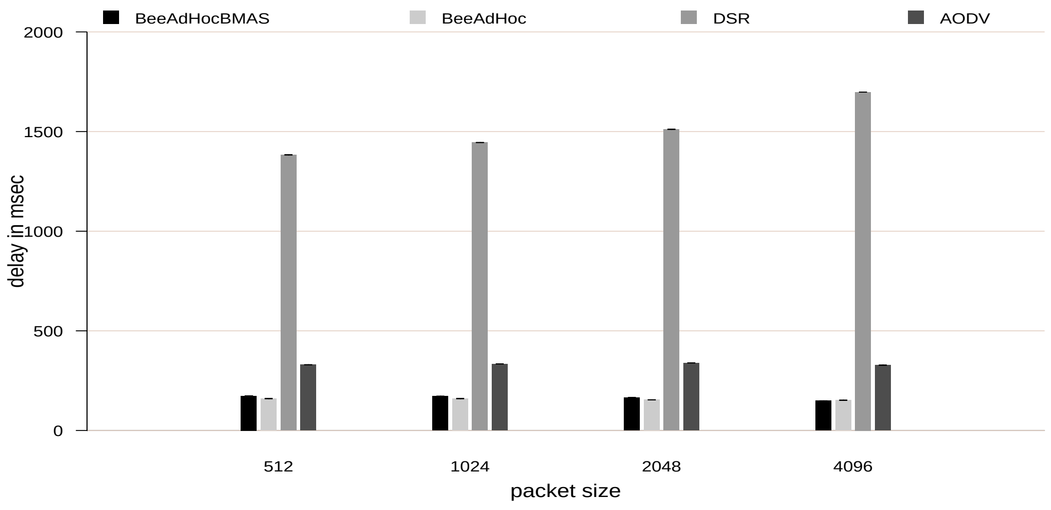


Figure 5.13.: Delay results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x)

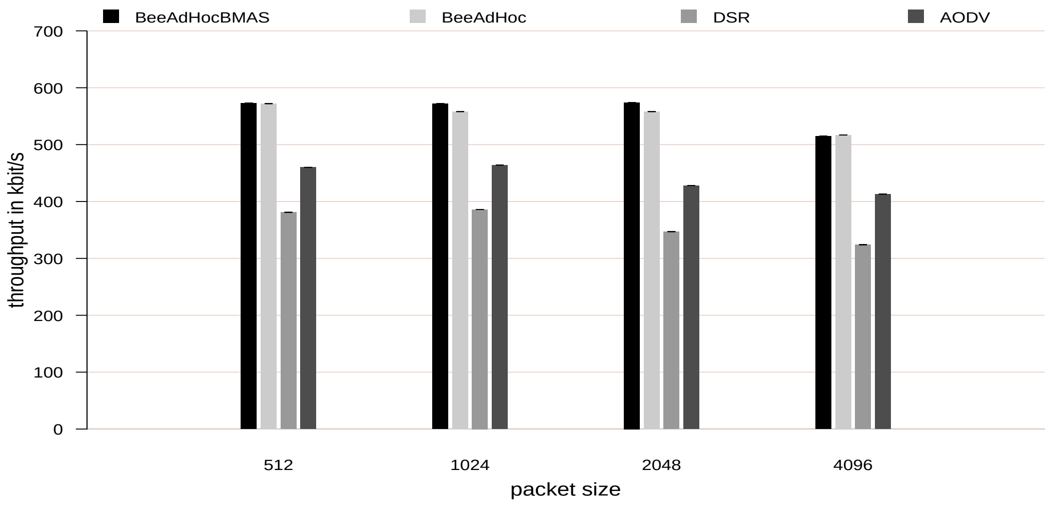


Figure 5.14.: Throughput results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x)

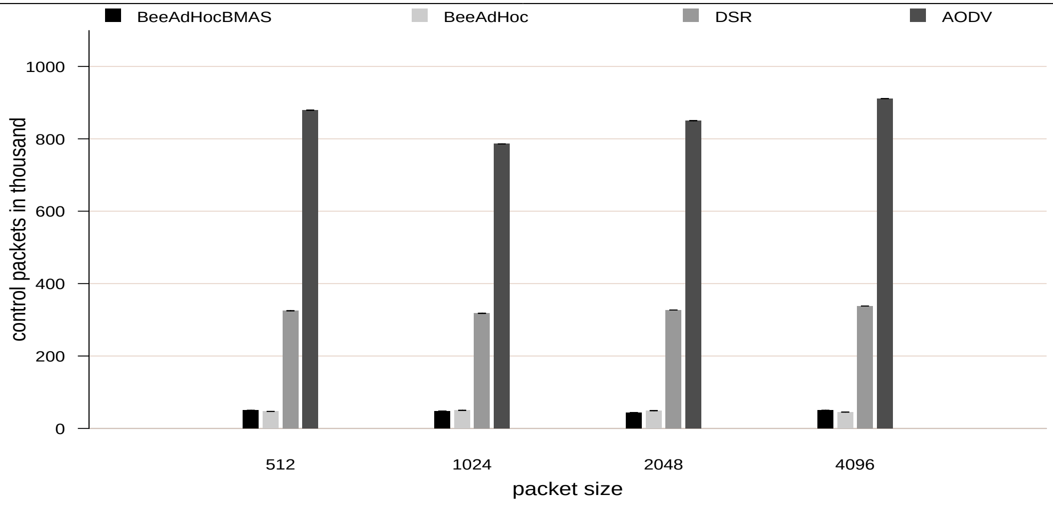


Figure 5.15.: Control packets results depending on packet size (from runs 3+x, 10+x, 11+x, 12+x)

5.6. Testing the Influence of Increasing Number of Nodes

We expected that energy consumption, delay and the amount of control packets increase and the throughput decreases. All these expectations were confirmed (see figures 5.16, 5.17, 5.18, 5.19).

As one can see BeeAdHocBMAS had always better energy, delay and throughput values than BeeAdHoc whereas BeeAdHoc needed less control packets. BeeAdHocBMAS had always the best delay. DSR and AODV needed 3 times less energy and they had twice a better throughput than BeeAdHoc or BeeAdHocBMAS. BeeAdHoc and BeeAdHocBMAS become better than DSR and AODV with increasing number of nodes.

5.7. Summary

Most simulation runs with 50 nodes yield equally good results for BeeAdHoc and BeeAdHocBMAS. BeeAdHocBMAS needed less energy but had a slightly worse delay. This shows that we can leave out the `FORAGER_DANCETIME` parameter as it has been described in section 1.

In the runs with 100 nodes BeeAdHocBMAS achieved smaller delay and higher throughput while it needed less energy than BeeAdHoc.

BeeAdHocBMAS is a really improvement if we have short pause times. In this case BeeAdHoc achieved significantly shorter delay times while it consumed half of energy used by BeeAdHoc. This shows that BeeAdHocBMAS was improved for a better adap-

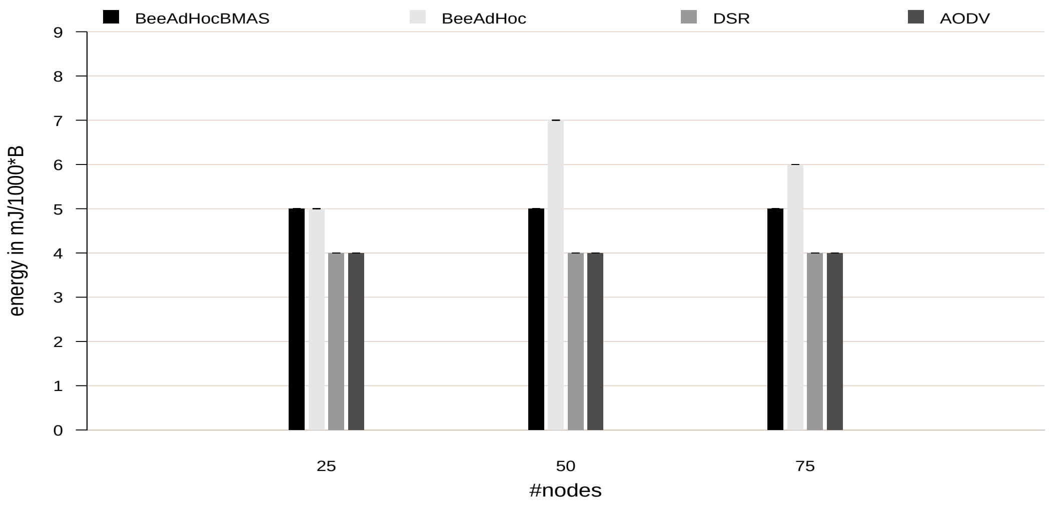


Figure 5.16.: Energy results depending on node number (from runs 9+x, 13+x, 14+x, 29+x)

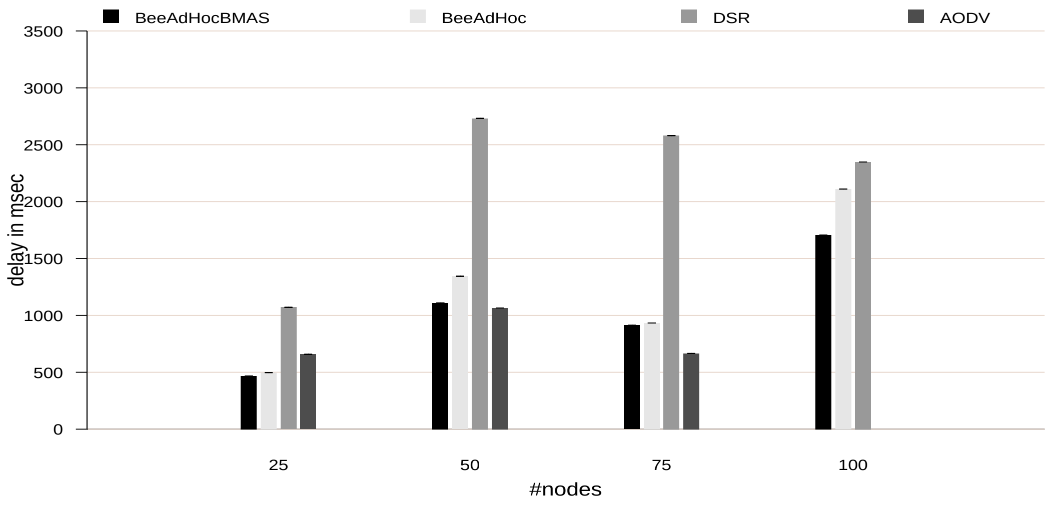


Figure 5.17.: Delay results depending on node number (from runs 9+x, 13+x, 14+x, 29+x)

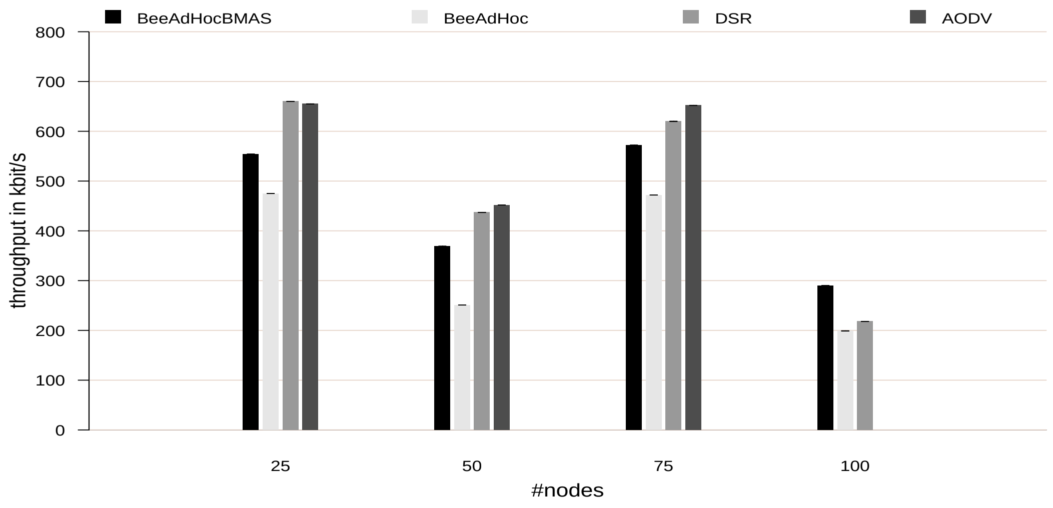


Figure 5.18.: Throughput results depending on node number (from runs 9+x, 13+x, 14+x, 29+x)

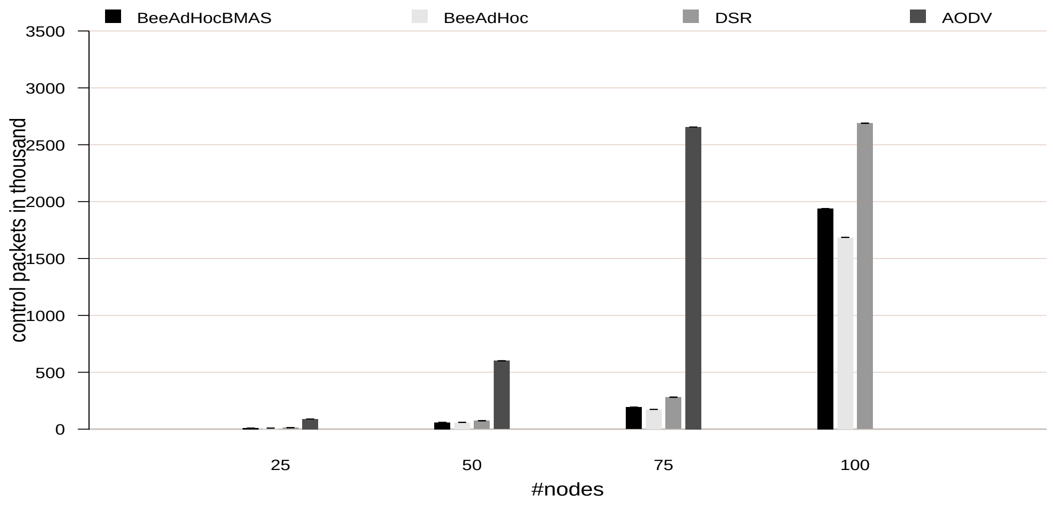


Figure 5.19.: Control packets results depending on node number (from runs 9+x, 13+x, 14+x, 29+x)

tion to mobility successfully.

6. Scalability Model Of BeeAdHoc

6.1. Introduction

This section will provide the motivation for our scalability features. After this there will be present the idea in pseudo code form so that it could be reimplemented on any simulation system. We will show snap shots of the source code where it is necessary for clarity.

6.2. Motivation

Starting position was the BeeAdHoc routing algorithm developed by the project group in 2004 [WFB+04]. In comparison to other popular protocols like AODV, DSR or DSDV it shows good performance in small networks with 50 nodes. But now our task was to modify the BeeAdHoc protocol such that it could scale in networks with 1000 nodes, i.e. without any significant degradation.

The reason for scalability is due to the fact that BeeAdHoc uses source routing. In this technique the complete route is packed in an IP header such that BeeAdHoc is only capable of reaching host within a radius of 9 hops because an IPv4 header could have a maximum of 10 IP addresses in the header field where the source route is stored.

6.3. Getting BeeAdHoc Scalable:Idea

6.3.1. Starting Point

We will describe our initial work briefly. Main aspects of routing in BeeAdHoc were these few points:

- destinations are found by sending out scout packets with a TTL parameter in the IP header by utilizing the expanding ring search technique
- every node appends its own IP address in the source route in the scout while the scout is exploring the network

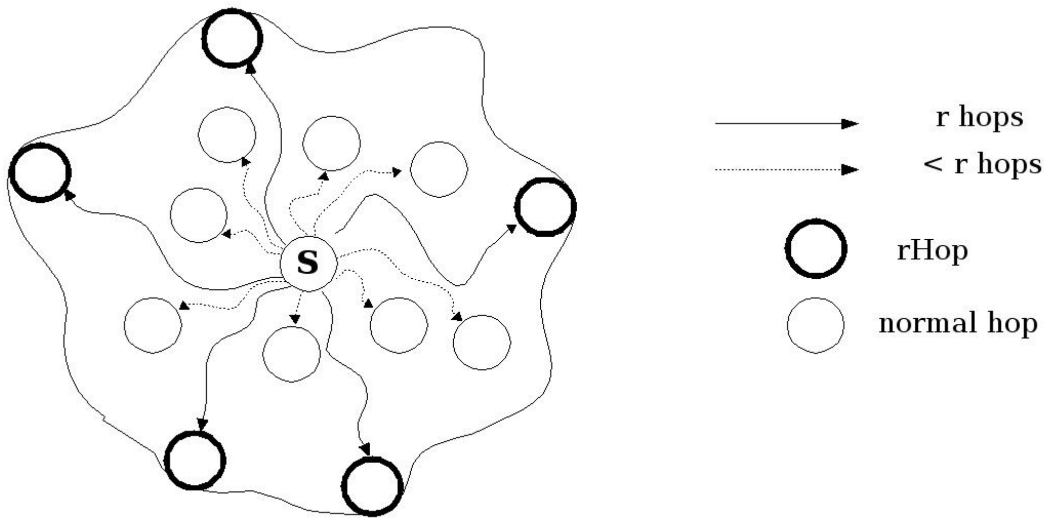


Figure 6.1.: Situation Before BeeAdHoc

- when a destination is found by a scout, the scout is sent back to its source node. In this way, a node can find a route to any destination

6.3.2. The Idea

Introduction

Our aim is to develop an extended BeeAdHoc algorithm while holding the main features of older BeeAdHoc version. We started our work with a BeeAdHoc version that delivered good results in a environment of ns2.

We thought about two approaches: The first option was storing the route in the data field which has unlimited capacity. The second option was switching to next hop routing. Both approaches have disadvantages. The first approach will require a significantly larger IP packet to be sent. The second approach is more cumbersome because of its fundamental shift in. Thus we would have to reimplement BeeAdHoc from scratch.

The simple idea is BeeAdHoc would be a good ad hoc routing algorithm if it could store routes of arbitrary length by simply replacing each ten hops the header with the next ones. But we do this swapping only every r-th hop which is a combination of next hop and source routing that leads to a hybrid routing algorithm. So the main idea is to let a packet traverse r hops with source routing and then this node decides which are the next r hops to be inserted into the source route. The r-th hop deciding which

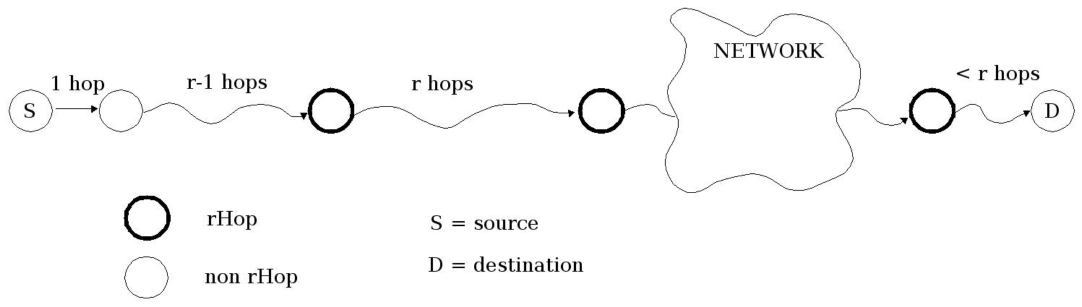


Figure 6.2.: Choice Of rHops

next r hops will be used are determined by starting to count from the source node which launched the scout (see figure 6.1 for the right way of counting). We call this Next R Hop Routing for a number R which denotes the radius and in our case it is 7 due to every 7th hop has to know which next 7 hops to take. In the following sections *src*, *dest*, *rte*, TTL,*id* will describe the values of the fields of the actual considered scout or forager .

If we look at figure 6.2 we will see the situation with the old BeeAdHoc version. S is only capable to reach the nodes in the bordered area. This are the nodes being in the 9 hop radius of S. With the idea described above, the border nodes get next rHops and will store next hop tables to reach the whole network.

This approach will help us in not modifying the parts like initiating scouts, scout timeout, forager creation. In this way we concentrate just on the scalability aspect. In the next few sections we will describe which is the extended features for launching scouts and foragers.

Scouting

A scout is processed for first seven hops as before. The small difference in the beginning is that two new fields *asrc* and *adest* get the values of the source node and the destination node. So each hop inserts its IP in the route in the scout and then will broadcast it again. The scout id is not modified thus a node will pass a unique scout only once. When no route is found within seven hop radius the initiating node will raise the TTL to greater than seven at some time. Thus all nodes that lie either at seven hop or a multiple of it are assigned a special function. We call such nodes as rHop. A scout will store its route until this node in a routing table for destination *asrc* while it is scouting. Furthermore it will empty the header and then insert itself at the beginning of route.

This action will be repeated until the destination is reached.

When the destination is reached it reverses the route, flips *asrc* and *adest* and broadcast the bee such that the bee now can reach the last rHop on the route. The rHop will save the route as a route to the field *asrc* of the packet (this will be the forward route: source - destination) and cause he knows the way back to *adest* (note that this is the source which was initiating the scout) he replaces the route by the route to the next rHop on the way back to *adest*.

Finally the scout will reach its source node and the source node has a route to the next rHop node towards the destination.

Forager Passing

Foragers are handled in a similar fashion as scouts. A source *src* having a forager to destination *dest* will know how to reach the next rHop on the way to *dest* (this information is stored in the forager's source header). So it will pack the forager with the data to send, and let the forager take its route to the next rHop via the source route. The rHop which the forager is sent to knows the next r hops on the way to *dest* and will write them into the route field of the forager and so on.

When a forager reaches at the destination its route is reversed and thus this forager is ready to fly back the same route.

Every rHop maintains a timer mechanism to keep up-to-date routes. Every rHop will not let it take too old routes.

6.4. BeeAdHoc Scalable: Model

6.4.1. Definitions

We will shortly explain the software architecture of our scalable model.

In order to understand scalability extensions made to BeeAdHoc one has to understand the difference between normal hops and rHops. rHop denotes a node which is r hops away from a given source node. The classification of rHop is a bit complicated due to route discovery algorithm. We consider a scout sent by node A to find a route to destination B. When it comes back to A and it recruits foragers the situation is clear: next rHop node is the eighth IP address in source header. However, the situation is vague if we consider the destination node B. In this case every r-th node might not necessarily be a rHop. The decision to select a node as rHop is taken in the scouting phase. Thus one has to start counting from the source node of a scout.

The name rHop is derived from the radius which should imply that only r hops can be covered without using the "special" nodes (rHops). The variable *radius* expresses the radius around a given node which can be covered without using rHops. Thus the selection of RADIUS will influence the rHop choice of the algorithm.

asrc, *adest*, *rte* and TTL will be the variables stored in the IP header.

6.4.2. Introduction

This section will describe the scouting and foraging mechanism in BeeAdHoc. We will discuss the behaviour through examples.

6.4.3. Information Carried By Scouts

We decided that it would be enough to save only partial routes in the header, therefore, we removed some information from the header. As a result store only the next 8 hops in the header. So the information about the source and destination of packets cannot be encoded in the new header.

Thus we have to add two new fields in the scouts and foragers. The new attributes are called *asrc* and *adest*. In *asrc* the source of the packet is stored and in *adest* the destination of the packet is stored.

For foragers the meaning of this attribute should be clear. A forager is on a route from A to B, then A is *asrc* and B is *adest*.

But the same is not true for scouts. They are sent from A to B and then from B to A. So on the way from A to B *asrc* is A and *adest* is B. On the return path from B to A, *asrc* is B and *adest* is A. For scouts scouting for a destination D the *dest* attribute stays at B the whole time.

6.4.4. Launching A Scout

Launching of scouts hasn't really changed. Thus the launching phase, the expanding ring search and the id mechanism haven't changed at all. So the first logical step of creating a route consists of a launching a scout which is similar to the old BeeAdHoc.

6.4.5. Forwarding A Scouting Scout

We have made significant changes in forwarding the scout algorithm.

Assume A has sent a scout to *dest* B. Now we have to consider two cases. The route stored in the header has size

1. < RADIUS - 1
2. = RADIUS - 1

Let us consider the first case. We do not need a rHop in this case. Thus this case is very similar to the old scouting algorithm. So a host decreases the TTL, inserts its IP address and broadcast the scout.

The second case is more challenging. The actual route stored in the header has size of (RADIUS -1) in which can only one IP address be inserted in the source header. Once the address of current node is inserted, then the header has reached its limit. This means that this hop is an rHop because we cannot increase the length of the source header.

In this case a scout with source *src* reached the node on its way for destination *dest.* Therefore, we can save the route *rte* (consisting of RADIUS hops) in the source header as next-r-hop-route to source *src.* Thus the pair *(src, rte)* is stored in the "next hop list". In this the current node can communicate with *src* over *rte.*At the end of route *rte* there is either *src* or a rHop which knows the next rHop towards *src.*

So we build the route for the scouts or foragers in a reverse direction.

6.4.6. A Scout At Its Destination

The above two scenarios can recursively happen: r times case one, one time case two and so on. If a path exists between *src* and *dest* the scout will reach its destination *dest.*

We already know that reverse route to rHop can be easily established from source header. A destination node just takes the following options.

- reverse its route
- flip *asrc* and *adest*
- set position to *rte.size()-1*
- send it out

6.4.7. Forwarding A Scout On Its Way Home

On the way back there are two main cases:

- position = 0
- position > 0

If position = 0 then either the hop is the source of the scout or it will be a rHop knowing with which route to replace *rte* such that the scout will reach its source. If it is a rHop it also stores the pair *(asrc, rte)* in its next hop table and in this way forwards a scout towards its source. If position > 0 then the node only have to transmit the scout to the next hop stored in the source route. So this will ensure that the scout takes the same route on its backward journey and reaches its source.

6.4.8. A Scout At Its Home

The source node takes the following actions

- flip *asrc* and *adest*
- reverse the *rte*
- convert it to a forager

Now the recruited forager knows a route to *adest* or the route to next rHop node which will then forward it to the destination *adest*.

6.4.9. Information Carried By Foragers

The two new proposed attributes only distinguish the forward or backward journey of a scout. However, there attributes are essential to forward foragers to next rHop or destination node. The forager uses *adest* field to map it to its next route and there it might still find another route to the destination if it is more than r hops away.

6.4.10. Initiation A Forager

We simply search for an available forager for the destination *adest* and if it exists then the data packet is given to it for transportation.

6.4.11. Forwarding A Forager

Here again we have the two cases. Either the receiving node is not a rHop then in case of it will forward it to the next hop in the source header. Otherwise a rHop node, it will replace the route for *adest*, stored in its routing tables, in the source header of a forager.

Thus if this hop is not a rHop it will only send the packet to hop *rte[position-1]*.

If it's a rHop it will replace the route in the header by the route stored in its next hop list under the key *adest* and will send the forager out.

6.4.12. A Forager At Its Destination

The route in the forager is reversed, *asrc* and *adest* will be switched. Then the data packet is recovered and the forager is added to the corresponding destination list.

6.5. Getting BeeAdHoc Scalable: Further Ideas

We could not explore other options due to time constraints.

The next hop mechanism works properly so far but could be extended and improved by connecting it more closely with the BeeAdHoc agent model. So next step to take would be identifying route not only by the route of r hops to the next rHop which they take but also by the rating which the route got on its last utilization. So the pair (source route, rating) could identify a route more clearly than it is the case until now. So, for example, a forager at a source node A has not the alternative in taking two different routes using the same rHop because a route a forager takes is only determined by the destination it will fly on the rHops side.

6.6. Implementation Of The Scalability Model In NS2

6.6.1. Introduction

This section will describe the implementation of our scalability model for BeeAdHoc in ns2. We provide an overview of the implementation of our algorithm. We, however, will describe the most important features for the sake of brevity.

Figures 6.3 and 6.4 show in a flow chart the way foragers or scouts are handled.

6.6.2. Scouting

Listing 37 shows the function Entrance::letIn(Scout* bee) which is called every time a scout arrives in the hive.

First we check if a replica of this scout has already passed this node. This happens by calling the function scoutListContains().

Then it is checked if the scout has reached the size RADIUS - 1 then it is safe to insert the node address (itself) into the route without violating the RADIUS limit. If it is so, the node pushes its address in the front of the route and saves the route into the

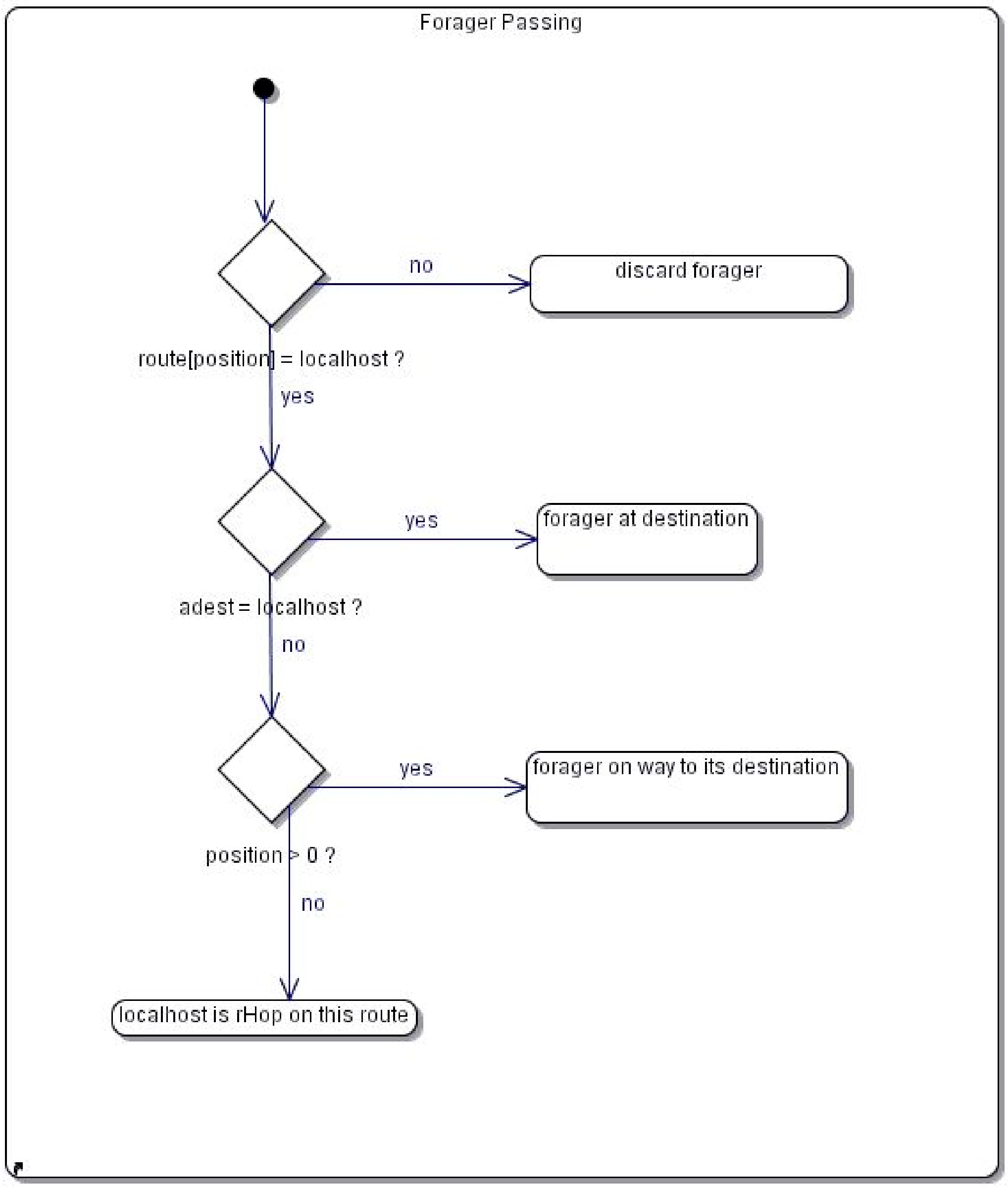


Figure 6.3.: BeeAdHoc: Forager Passing

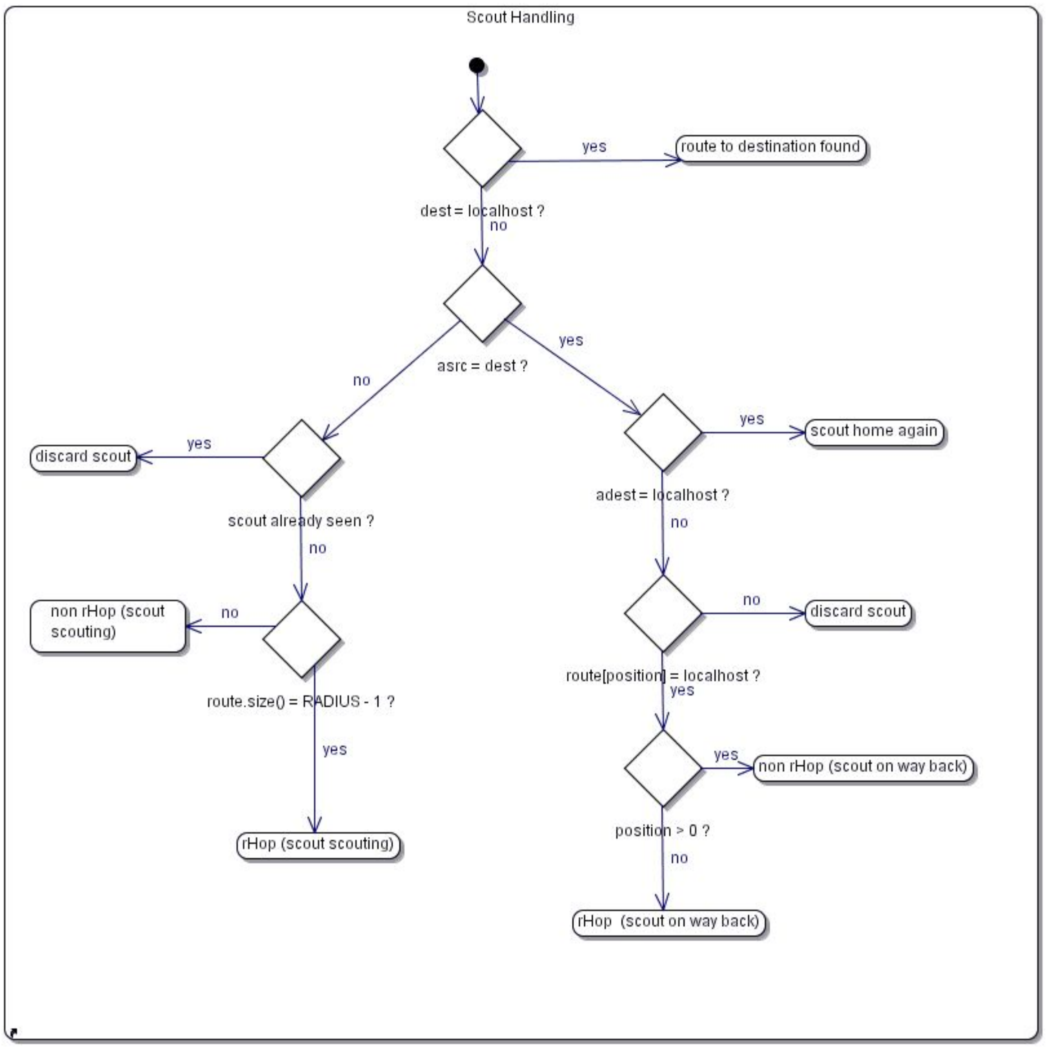


Figure 6.4.: BeeAdHoc: Scout Handling

nexthop-list. The method saveRoute() will take care of storing the route in the right order itself.

After the route is saved the route in the header is deleted and the space is available for a new route of seven hops. So the node inserts its IP address at the beginning of the route and broadcasts the scout again.

Second case is that this is a normal hop which will only broadcast the scout.

However, if the number of hops is less than RADIUS-1 then the scout is again broadcasted.

```
void Entrance::letIn(Scout* bee) {

...
        ...

        else if(!scoutListContains(bee)) {
                        if (bee->route.size() == beehive->RADIUS - 1)
                           {
                                bee->route.push_front(localaddress);
                                bee->ttl--;
                                if(bee->ttl <= 0) {
                                        beehive->cleanupScout(bee);
                                        return;
                                }
                                if (beehive->SCOUT_CACHING)
                                        cacheScout(bee);
                                saveRoute(bee->asrc, bee->route);
                                bee->route.clear();
                                bee->route.push_front(localaddress);
                                beehive->broadcastBee(bee);
                        }else{
                                bee->route.push_front(localaddress);
                                bee->ttl--;
                                if(bee->ttl <= 0) {
                                        beehive->cleanupScout(bee);
                                        return;
                                }
                                beehive->broadcastBee(bee);
                        }
        }
        else {
```

```
                    beehive->cleanupScout(bee);
          }

          ...
...
}
```

Listing 6.1: Scouting Phase

6.6.3. Scout Arrival: Destination/Source

Next case to be considered is if the scout has reached its destination or is at its home again (listing 6.2).

In both cases the insert their IP addresses in front of route, *src* and *adest* are flipped and the route is reversed. When a scout reaches the destination its TTL is set to infinity and is sent out and when it reaches its source node the scout is sent to the packing floor which will create foragers for this scout.

```
void Entrance::letIn(Scout* bee) {
...

          if(bee->dest == localaddress) {
                    bee->route.push_front(localaddress);
                    reverseRoute(bee);
                    bee->adest = bee->asrc;
                    bee->asrc = bee->dest;
                    bee->ttl = beehive->ttlInf;
                    bee->position = bee->route.size() - 1;
                    bee->position--;
                    beehive->sendBee(bee);
                    return;
          }

          else if ((bee->asrc == bee->dest)) {
                    if (bee->adest == localaddress){
                              bee->adest = bee->asrc;
                              bee->asrc = localaddress;
                              reverseRoute(bee);
                              packingFloor->watchPackingFloor(bee);
                              return;
                    }else{
```

```
          ...
...
}
```

Listing 6.2: Scout: Source Destination Arrival

6.6.4. Scout On Way Back

```
void Entrance::letIn(Scout* bee) {
...
          ...

                    if (bee->route[bee->position] != localaddress){
                              beehive->cleanupScout(bee);
                              return;
                    }
                    else if (bee->position == 0){
                              saveRoute(bee->asrc, bee->route);
                              bee->route = getRoute(bee->adest);
                              if(bee->route.empty()){
                                        beehive->cleanupScout(bee);
                                        return;
                              }
                              bee->position = bee->route.size() - 1;
                              bee->position--;
                              beehive->sendBee(bee);
                    }else{
                              if (bee->position > 0){
                                        bee->position--;
                                        beehive->sendBee(bee);
                                        return;
                              }else{
                                        beehive->cleanupScout(bee);
                                        return;
                              }
                    }

          ...
...
}
```

Listing 6.3: Scout On Way Home

Once a node receives the returning scout then it simply stores routes if it is a rHop(position = 0). Otherwise it is forwarded to the next hop as indicated in the source header.

The function getRoute returns the route to the next rHop towards *adest*.

6.6.5. Forager Passing

The mechanisms used here are already described above. If a forager has reached its destination then *asrc* and *adest* are flipped and the forager is sent to the packingFloor. The route is reversed there and then stored in the appropriate lists.

A forager is simply sent to the next hop stored in the route. This is done by the method letOut(), if this is not a rHop (position != 0). Otherwise the node replaces the route with the route to the next rHop and then retransmits it.

```
void Entrance::letIn(Forager* bee) {
        if (bee->route[bee->position] != localaddress){
                beehive->dropForager(bee);
        }
        else if (bee->adest == localaddress) {
                updateThroughput(bee->packetSize);
                updateInfo(bee);
                bee->lastFlight = beehive->simTime();
                bee->adest = bee->asrc;
                bee->asrc = localaddress;
                packingFloor->receive(bee);
        }
        else {
                if(bee->position > 0) {
                        letOut(bee);
                }else {
                        if(bee->position == 0){
                                updateLastVisit(bee->asrc, bee->route
                                    );
                                bee->route = getRoute(bee->adest);
                                if (!bee->route.empty()){
                                        bee->position = bee->route.
                                            size()-1;
                                        letOut(bee);
                                }else{
                                        beehive->dropForager(bee);
                                }
                        }else{
                                beehive->dropForager(bee);
                        }
                }
        }
}
```

Listing 6.4: Forager Passing

6.6.6. Next Hop List

Routes are stored in a which could not get larger than the number of traffic nodes in the network. Thus it is bounded by number of nodes in the network.

The important aspect of this data structure is storing the routes. The order of nodes in the source route is reversed while saving them. This helps in retrieving the routes for data packets in the correct order.

```
        void Entrance::saveRoute(int dst, deque<int> route){
        bool exists = false;
        for (list<rHopEntry*>::iterator j = rHopList.begin(); j!=
            rHopList.end(); j++) {
                if ((*j)->dest == dst){
                        (*j)->route.clear();
                        for(int i=0; i < route.size(); i++)
                                (*j)->route.push_front(route[i]);
                        exists = true;
                }
        }
        if (exists == false){
                rHopEntry* entry = new rHopEntry();
                entry->dest = dst;
                entry->lastVisitation = beehive->simTime();
                for(int i=0; i < route.size(); i++)
                        entry->route.push_front(route[i]);
                rHopList.push_back(entry);
        }
}
```

Listing 6.5: Entrance::saveRoute()

7. Scalability Issues in MANETs

7.1. Simulation framework

7.1.1. Introduction

This chapter deals with the evaluation and simulation of our BeeAdHoc algorithm. The BeeHive algorithm was developed with the background to be used in Wireless networks with up to 200 nodes. The Beeadhoc algorithm in contrast is a reactive source routing algorithm and it consumes less energy as compared to existing state-of-theart routing algorithms because it utilizes less control packets to do routing and can be used in wireless networks up to 500 nodes. Since it takes enormous efforts and costs to evaluate such large networks in reality it was decided to do simulation and evaluation with a software simulator.

7.1.2. Considered Algorithms

We tested and compared the three MANET algorithms AODV, DSR and DSDV beside BeeAdhoc. This are the following:

AODV

The name is self-explanatory Adhoc On-demand Distance Vector routing protocol. Thus AODV is a next hop routing algorithm where every hop has its own routing table. This table is updated on demand. Let a source node S has a packet to be deliverd to destination node D; there will be the following sequence of events, if S has no next hop entry for destination D:

- S broadcasts a RouteRequest(RREQ) message to all its neighbors
- until destination D is reached every node receiving a RREQ message broadcasts it again to its neighbors
- when destination D receives the RREQ message for himself it sends a RouteReply (RREP) the way back the RREQ message has taken to reach it

- a intermediate node on the RREQ way to D can everytime complete the route and send a RREP back
- nodes on the way back of the RREP packet set the entries in the route table for the destination node of the RREP packet onto the node from which the RREP packet came
- if no RREP arrives at S in the timeout intervall it sends a new RREQ with a higher TTL value
- if a timeout for the RREQ packet occurs and the RREQ packet has a TTL of the whole network node D is set as unreachable Of course there are more details like sequence numbers, Route-Error packets (RERR),

DSR

The Dynamic Source Routing Algorithm (DSR) is also an on-demand routing protocol. The algorithm has similarites to the behaviour of AODV and is based on the concept of source routing. The route discovery process occurs as follows

- The source broadcasts a route request packet (RREQ) to all reachable nodes
- The reached nodes add their node IDs into the route header and forwards the packets to all its neighbour nodes.
- When the packet reaches the destination the header contains the complete route to the source, which is inserted in the route header of the RREP
- Every node checks the route header of the RREQ packet for its own node ID to avoid loops
- If the sources receives a route reply, it caches the source route and includes it in the header of each data packet. The accurate transmission of a packet is ensured by the intermediate nodes, which forward the packet according to the route specified in the header and also cache the route of the route header in their route cache. To guarantee the delivering acknowledgements are send from every node after receiving a data or When it does not receive an acknowledgement, it will make a resent. If there is a route error message, caused by the data link layer encountering a fatal transmission problem, the hop in error is removed from the nodes route cache and all routes containing this hop are cut at that point. Many optimizations for DSR passed on aggressive caching and analysis of topology information are incorporated into this scheme.

DSDV

In the opposite to the on-demand DSR and AODV protocols the DSDV algorithm is a table-driven protocol in which every node has it own routing table and the nodes exchange there information about active route via messages to update these tables. Each node maintains the following information in its routing table:

- next hop towards each destination
- a cost metric for each destination
- a destination sequence number that is created by the destination itself
- sequence number to avoid loops Every node periodically sends its routing table to its neighbour. Meanwhile the node increments and appends its sequence number. This sequence number will be attached to route entries created for this route.

7.1.3. Metrics

The algorithms were evaluated in terms of average end-to-end delay per packet and packet delivery ratio (i.e., the fraction of successfully delivered data packets). We now define the metrics which we used in the comparison of the algorithms.

Energy per user data

The total energy consumed, including the energy consumed by the control packets, to transport one kilobyte of data to its destination. This metric is minimum when the same number of bytes could be delivered at the destinations in less hops and with small number of control packets. This metric is also referred to as energy expenditure.

Success rate

The ratio between the number of packets successfully received by the application layer of a destination node and the number of packets originated at the application layer of each node for that destination. This parameter is also referred to as packet delivery ratio.

Delay

The difference between the time once the packet is received by the application layer of a destination node and the time when the packet was originated at the application layer of a source node. This definition takes care of the time that a packet has to wait at the source node while the route to its destination is to be found (reactive wait time).

We always report the 100th percentile of the delays distribution because it provides an insight on the spread of the delays which is an important criterion for quality of service (QoS) applications, in which all packets should arrive at the destination within an acceptable variance from the mean.

Throughput

If y number of bits are delivered within t time at a node then the throughput at the node could be defined as y/t . This definition assigns a higher throughput value to an algorithm that delivers the same number of y bits in a smaller time. This definition of throughput implicitly strikes a good balance between the number of packets delivered at a node and their delays.

7.1.4. Simulation environment

Previous tests

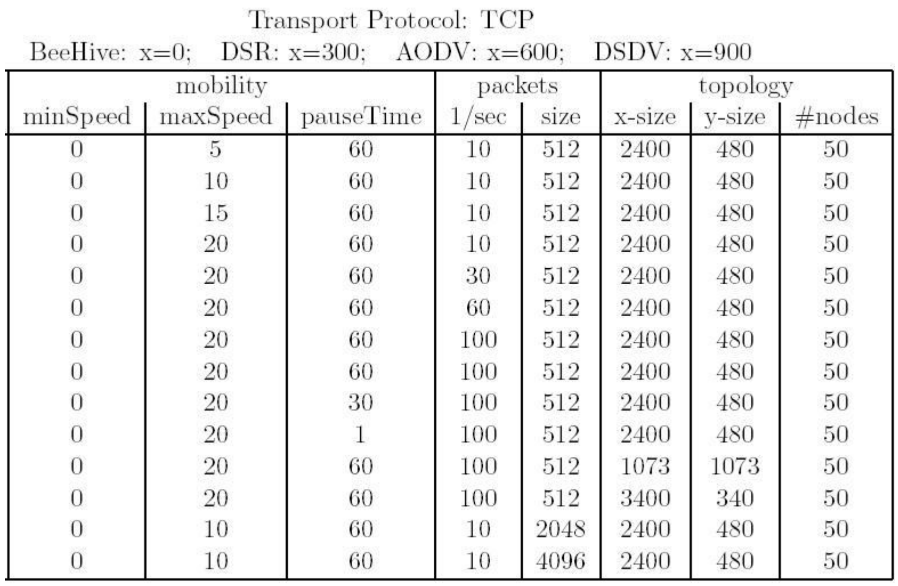
To compare all these algorithms with one other, a testing environment has to be created in which the starting position of the nodes and other conditions are the same for all algorithms. This could be done through creating a simulation environment in ns-2. But testing these algorithms for only one simulation case (or test case) can provide very specific results, to get more 'across the board' results. We created a number of test scenarios and repeated each scenario .The amount of simulated seconds was set to 1000 for every simulation run.

Each of the runs was repeated five times with different seeds, which result in a slightly changed order of events.These five results for each run were merged together to build an average case, which was then used for comparisons. We splitted our experiments on different machines for speeding up the evaluation process.

We used small shell scripts that looks like this:

Several other tests

In addition to the above-mentioned scenarios, we studied the experimental setup in which AntHocNet was evaluated. AntHocNet is also a routing algortihm for mobile ad hoc networks. networks. It is a hybrid algorithm, which combines reactive route setup with proactive route probing, maintenance and improvement. We used the same setup as was used in the scalability study of AntHocNet. We conducted the experiments with similar scenarios. In the first scenario, 100 nodes are randomly placed inside an area of 3000m x1000m. Each experiment is run for 900 seconds. Data traffic is generated by

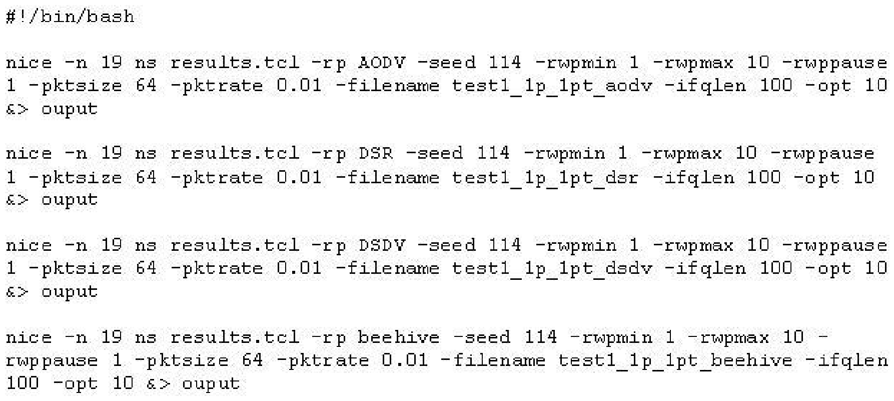


Transport Protocol: TCP

BeeHive: x=0; DSR: x=300; AODV: x=600; DSDV: x=900

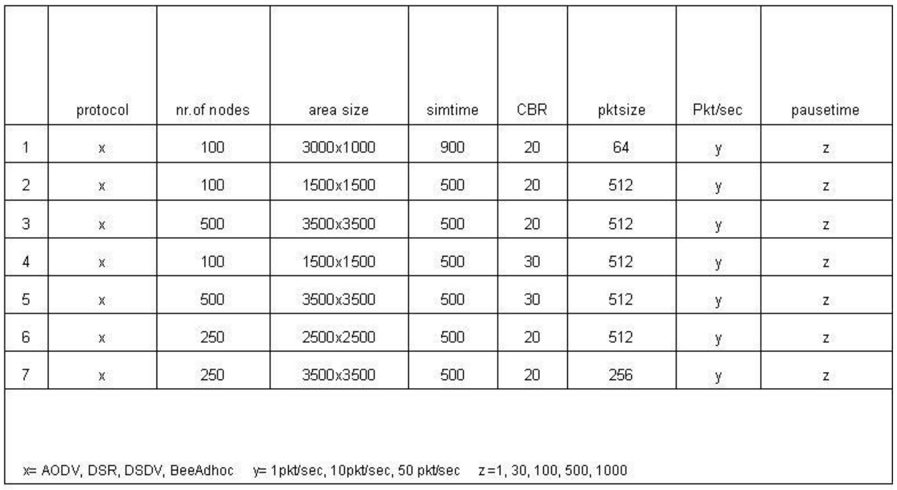
mobility			packets		topology		
minSpeed	maxSpeed	pauseTime	1/sec	size	x-size	y-size	#nodes
0	5	60	10	512	2400	480	50
0	10	60	10	512	2400	480	50
0	15	60	10	512	2400	480	50
0	20	60	10	512	2400	480	50
0	20	60	30	512	2400	480	50
0	20	60	60	512	2400	480	50
0	20	60	100	512	2400	480	50
0	20	60	100	512	2400	480	50
0	20	30	100	512	2400	480	50
0	20	1	100	512	2400	480	50
0	20	60	100	512	1073	1073	50
0	20	60	100	512	3400	340	50
0	10	60	10	2048	2400	480	50
0	10	60	10	4096	2400	480	50

Figure 7.1.: Overview of the tests in node 50



```
#!/bin/bash

nice -n 19 ns results.tcl -rp AODV -seed 114 -rwpmin 1 -rwpmax 10 -rwppause
1 -pktsize 64 -pktrate 0.01 -filename test1_1p_1pt_aodv -ifqlen 100 -opt 10
&> ouput

nice -n 19 ns results.tcl -rp DSR -seed 114 -rwpmin 1 -rwpmax 10 -rwppause
1 -pktsize 64 -pktrate 0.01 -filename test1_1p_1pt_dsr -ifqlen 100 -opt 10
&> ouput

nice -n 19 ns results.tcl -rp DSDV -seed 114 -rwpmin 1 -rwpmax 10 -rwppause
1 -pktsize 64 -pktrate 0.01 -filename test1_1p_1pt_dsdv -ifqlen 100 -opt 10
&> ouput

nice -n 19 ns results.tcl -rp beehive -seed 114 -rwpmin 1 -rwpmax 10 -
rwppause 1 -pktsize 64 -pktrate 0.01 -filename test1_1p_1pt_beehive -ifqlen
100 -opt 10 &> ouput
```

Figure 7.2.: example of a shell script



	protocol	nr.of nodes	area size	simtime	CBR	pktsize	Pkt/sec	pausetime
1	x	100	3000x1000	900	20	64	y	z
2	x	100	1500x1500	500	20	512	y	z
3	x	500	3500x3500	500	20	512	y	z
4	x	100	1500x1500	500	30	512	y	z
5	x	500	3500x3500	500	30	512	y	z
6	x	250	2500x2500	500	20	512	y	z
7	x	250	3500x3500	500	20	256	y	z

x= AODV, DSR, DSDV, BeeAdhoc y= 1pkt/sec, 10pkt/sec, 50 pkt/sec z=1, 30, 100, 500, 1000

20 constant bit rate (CBR) sources sending four 512-byte packets per second. For the second setting, we used the same setup as was used in the scalability study of AODV performed by Lee, Belding-Royer and Perkins . In this study, the number of nodes and the size of the simulation area are varied, the exact values used for the number of nodes and the size of the area are given in Table 1.

Other properties of the simulation setup are kept constant over the different test scenarios. The nodes move according to the random waypoint model, with a minimum speed of 0 m/s, a maximum speed of 10 m/s, and a pause time of 30 seconds. For each of the different settings of the parameter values, minimum three different problems were created, by choosing different seeds.

In our testings we considered also three parameters, which come to the fore. With these different parameters each several test was accomplished. These settings are the varied pause time, the number of the nodes and the different protocols. The used protocols are AODV, DSR, DSDV(which was left out in some cases due to errors) and BeeAdhoc. Then, the pause time is varied. Each node moves randomly to a specific destination, waits there for a time which is represented by pause time. Then it moves to another point, waits again and so on. We used therefore 5 different parameters which range from 1 second to 1000 seconds. Finally the number of the nodes modified. Since we have tested in the previus tests with 50 nodes, we tested here with 100, 250 and 500 nodes.

7.1.5. Testing results

7.2. Testing results

7.2.1. simulation runs

In order to make a good comparison of the tested algorithms, we do plot important performance values which are represent an average value of independent runs. The result files in the appendix could be interpreted as in the following:

```
------------------------------------------------------------------------------------
  Run-Nr   | Pause time    |Send rate  |           Topology          |Pktsize| Simtime
           |               |           |x-size |y-size  |Number nodes|       |
------------------------------------------------------------------------------------
  x +  c   |      y        |     z     |       a        |  n         |  s    |   st
------------------------------------------------------------------------------------

Where the values range are the following:

 x={0,200,400,600} BeeAdHoc=0, AODV=200, DSDV=400, DSR=600
 c={1..105}
 y={1,30,100,500,1000}
 z={1,10,50}
 a={3000x1000,1500x1500,3500x3500, 2500x2500}
 n={100,250,500}
 s={64,512}
 st={500, 900}
```

Figure 7.3.: Run configuration

7.2.2. Testing Energy behavior per user data

The energy consumption of each algorithm as function of varying pause time is shown in the figure 7.4. The figure 7.5 shows the energy consumption as we increase the size of the network.
The third graphic 7.7 shows the influence of node velocity towards the energy that has been used to deliver one kB of user data to the desired destination in average.
The three figures represent the energy consumption behavior which occur when the pause time, the number of nodes or the node velocity varies.

BeeAdHoc has the lowest energy consumption in the three cases. When the pause time increases, AODV and DSDV are consuming more than BeeAdHoc but less than DSR. DSR has the worst energy consumption among all of the tested algorithms.
When the number of nodes increases the energy consumption behavior of BeeAdHoc is similar to AODV. BeeAdHoc have the best values compared to AODV when the number of Nodes is 250 and 500. For the third case see figure 7.7, The best values are from DSDV, when the speed varies from 0 to 15 m/s and from 0 to 20 m/s The value of DSDV stays constant during the variation. The ones of BeeAdHoc and AODV decrease with the augmentation of the maximum speed. The value of DSR increase with the augmentation of the maximum speed.

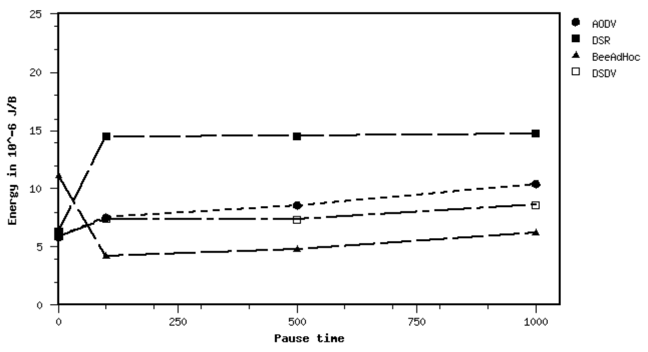


Figure 7.4.: Energy depending on pause time from runs x+1, x+2, x+3, x+4 x+5

7.2.3. Testing Delay behavior

The values of delays of each algorithm as we vary the pause time are shown in the figure 7.8. Similarly the delay values for different sizes of network are shown in figure 7.9
BeeAdHoc has the best delay values for the tested algorithms. The behavior of all

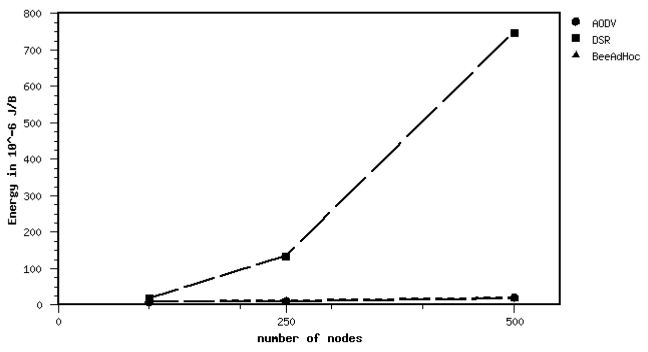


Figure 7.5.: Energy depending on number of nodes average values from runs x+1, x+16, x+31, x+46, x+61, x+76

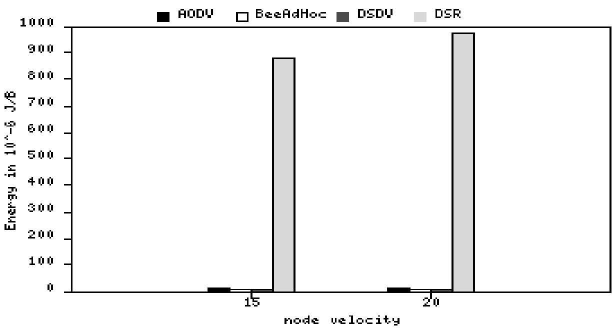


Figure 7.6.: Energy depending on node velocity average values of all runs

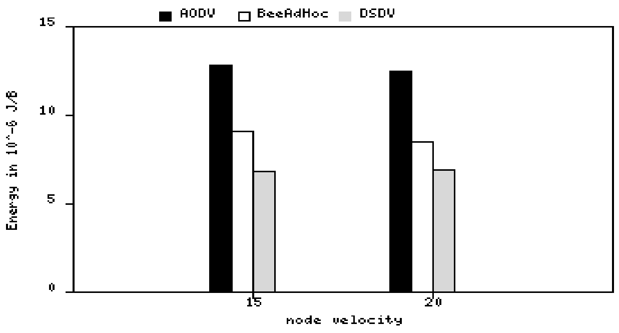


Figure 7.7.: Energy depending on node velocity just for AODV, BeeAdHoc and DSDV

algorithms is sam i.e it decrease with an increase of the pause time.DSR has the worst delay values. AODV has relatively better delay. Behavior of algorithms for different size of network is the same. The delay is high at 100 node network, then it decreases at 250 nodes network and then slightly increases again at 500 nodes network. The third figure shown in figure 7.10 shows the influence of varied speed on the delay of each algorithm. The best values are from BeeAdHoc. the values of BeeAdHoc stay constant with the variation of the maximum speed. The second best values are from DSDV and the third from AODV. Both Algorithm have also constant values with variation of the maximum speed. The worse values compared to all tested algorithms are the ones of DSR, which increase with the variation of the maximum speed.

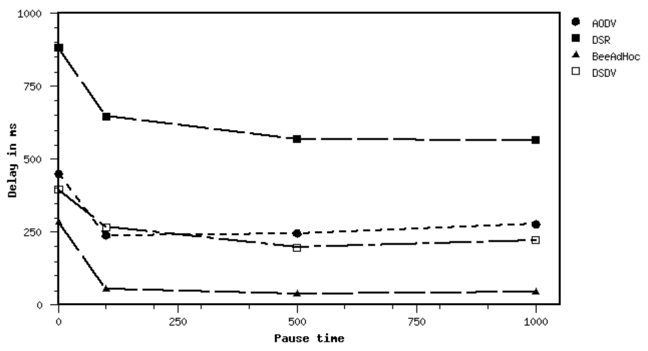


Figure 7.8.: Delay depending on pause time from runs x+6, x+7, x+8, x+9 x+10

7.2.4. Testing Throughput behavior

Figure 7.11 shows a comparison of the throughput of AODV, BeeAdHoc, DSDV and DSR as we vary the pause time and figure 7.12 shows the the throughput of each algorithm as

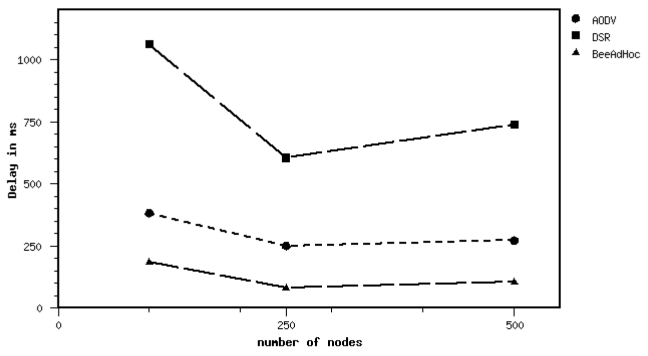


Figure 7.9.: Delay depending on number of nodes average values from runs x+5, x+20, x+35, x+50, x+65, x+80

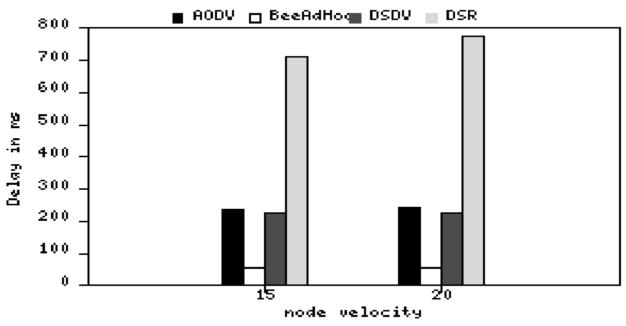


Figure 7.10.: Delay depending on node velocity average values of all runs

we increase the size of the network. The throughput of BeeAdHoc is the best among all of the algorithms. Then comes DSR and AODV has the worst Throughput. The behavior of throughput is same: when pause time increases up to 100s the throughput of tested algorithms increases hen the pause time is between 100s and 500s the throughput stabilize for BeeAdHoc and AODV but decrease for DSR. But DSR throughput is still better than AODV, and when the pause time is more than 500s, the value of the throughput decreases for each tested algorithm.
The throughput behavior is same for different network sizes: BeeAdHoc has the best throughput and DSR the worst.
The third curve see figure 7.13 shows the run of the curves of the tested algorithms depending on the variation of the node speed. Also here BeeAdHoc has noticeable nice and the best values. The second best are from DSDV, the third are from AODV and the last from DSR. For all algorithms, the run of the curve is almost constant with the variation of the speed.

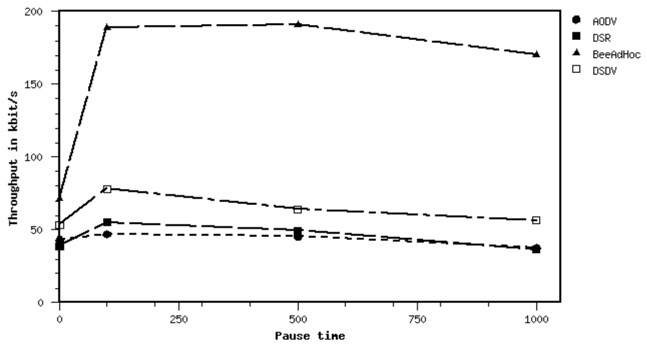


Figure 7.11.: Throughput depending on pause time from runs x+6, x+7, x+8, x+9 x+10

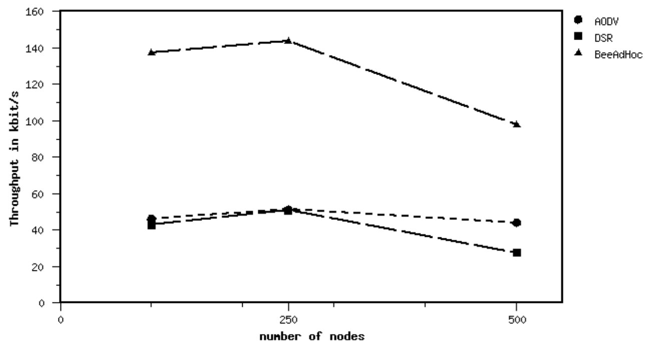


Figure 7.12.: Throughput depending on number of nodes average values from runs x+2, x+17, x+33, x+48, x+63, x+78

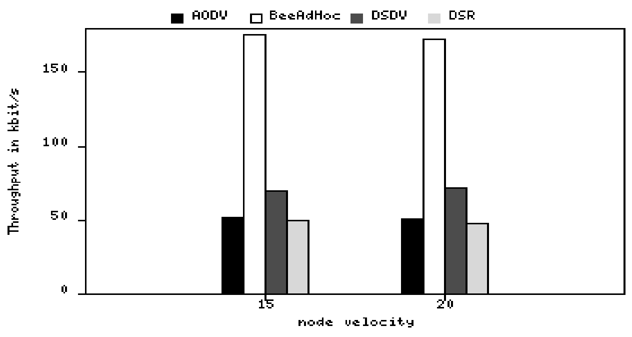


Figure 7.13.: Throughput depending on node velocity average values of all runs

7.2.5. Testing control packet behavior

Figure 7.14 shows the number of control packet sent by each protocol as we vary the pause time.
BeeAdHoc sends the smallest number of control packet as shown in the figure 7.14. The reason for less consumption of energy in BeeAdHoc . The number of packets sent by BeeAdHoc algorithm is also constant with the variation of the pause time. For AODV the number of control packet increase when the pause time is varied between 1 s and 100s and when the pause time remain constant. For the DSR we can see a drastically increase when the pause time is varied from 1s to 100s and when the pause is more than 100s the number of control packets sent by DSR decreases.

The figure 7.15 shows the throughput curves of the three tested algorithm as we vary the network size. We can also notice here BeeAdHoc launches the least number of control packets. The number of control packets for DSR increase dramatically.
The last graphic shows the influence of the variation of speed on control packet send by each algorithm. BeeAdHoc is also the best here follow by DSDV. The third best values are from AODV. The last are from DSR. The run of the curves is almost the same (constant with the variation of the speed) for each algorithm without DSR. For DSR, the curve decrease with the augmentation of the maximum speed.

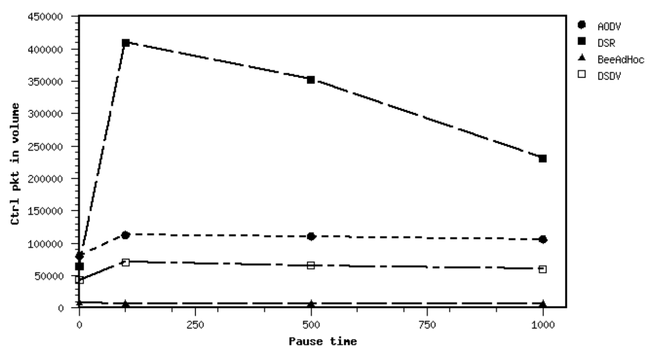


Figure 7.14.: control packet depending on pause time from runs x+1, x+2, x+3, x+4 x+5

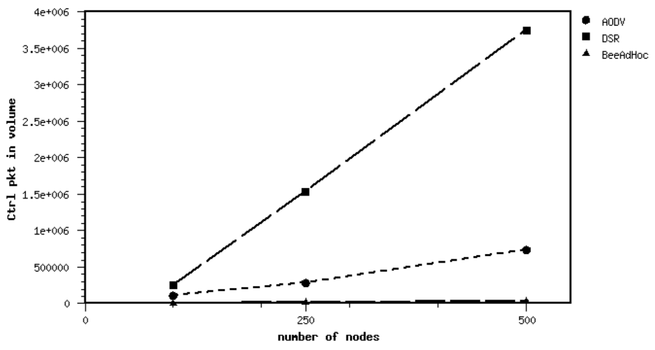


Figure 7.15.: Control packet depending on number of nodes average values from runs x+11,x+26,x+41,x+56,x+71,x+86

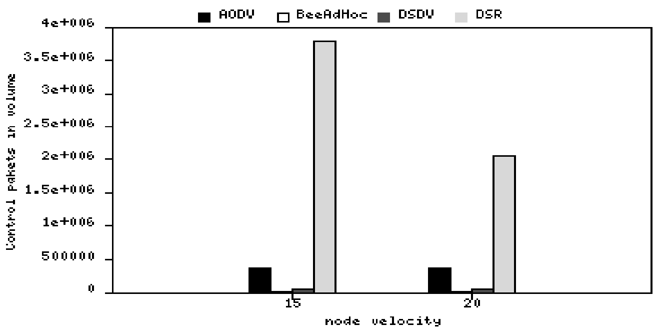


Figure 7.16.: control packet depending on node velocity average values of all runs

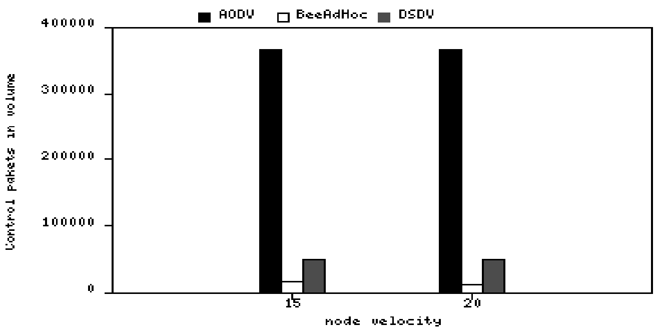


Figure 7.17.: control packet depending on node velocity average values of all runs

7.2.6. Testing the packet delivery ratio behavior

The Packet delivery ratio is defined as the percentage of received packets, relative to the total number of packets actually generated. The following graphs show the influence of the pause time variation and the number of nodes variation on the packet delivery ratio of each tested algorithm.
Figure 7.18 shows the Packet delivery ratio as we vary the pause time. We can observe on these curves that packet delivery ratio of all the tested algorithms is between 90 percent and 100 percent.
The packet delivery ratio of BeeAdHoc when the pause time is small than 100s is the lowest, DSR has the best value in this range. When the pause time is between 100s and 500s, BeeAdHoc has the best values and then it maintains it.
Figure 7.19 shows the packet delivery ratio as we increase the network size. We can observe a similar behavior for the three protocols:
Packet delivery ratio drops in 250 nodes network and then again increases in 500 nodes network
The last graphic shows the run of the curve of each tested algorithm depending on the node velocity. We can see that DSDV values are the best and they are constant with the variation. The values of DSR and BeeAdHoc varies when the maximum speed is increased. BeeAdHoc increases and DSR decreases. The worse value are the ones of AODV and they increase slightly when the maximum speed is increase from 15 to 20 m/s.

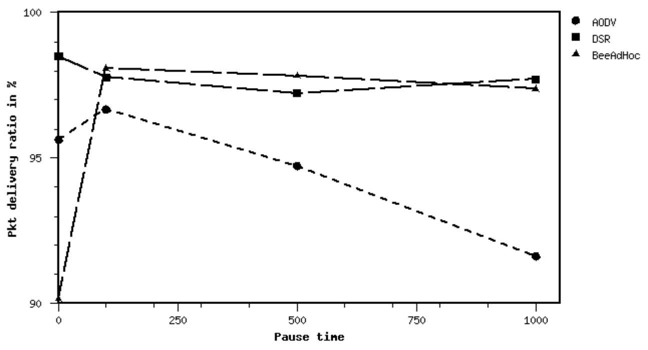


Figure 7.18.: Packet delivery ratio depending on pause time from runs x+51, x+52, x+53, x+54 x+55

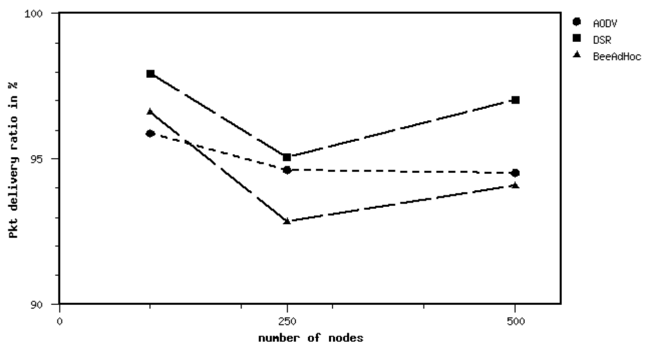


Figure 7.19.: Packet delivery ratio on number of nodes average values from runs x+4, x+19, x+34, x+49, x+64, x+79

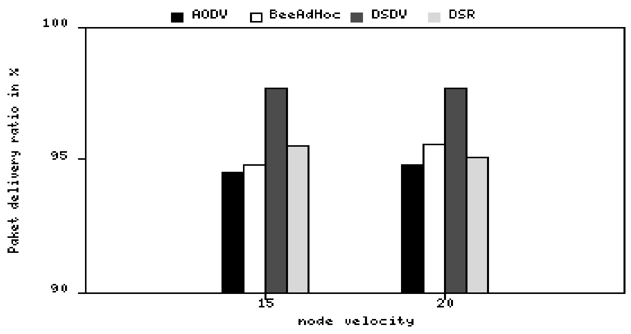


Figure 7.20.: Packet delivery ratio depending on node velocity average values of all runs

8. Security in MANETs

Abstract

Mobile Ad Hoc Networks (MANETs) have become an important class of networks in the recent years because of the rapid developments in wireless communication.
But one of the serious problems is that MANETs are vulnerable to attacks due to wireless medium, dynamically changing network topology and the lack of centralized monitoring.
The absence of a central control unit that could provide authenticity mechanisms makes the task of identifying attacks and malicious nodes very difficult.
So securing MANETs is just as important, if not more, as securing traditional wired networks. Nevertheless, solutions for fixed networks are generally not suitable for wireless ones, because Ad Hoc Networks have their own vulnerabilities.
Secure Ad Hoc Network routing protocols are difficult to design, due to dynamic nature of Ad Hoc Networks. In addition the nodes need to operate efficiently under the constraint of resources: network bandwidth, CPU processing, memory and limited battery capacity.
Existing insecure Ad Hoc Network routing protocols are highly optimized to distribute new routing information quickly as conditions change. This requires more frequent information exchange between nodes than in traditional networks.

Introduction

The use of a wireless channel renders an Ad Hoc Network vulnerable to malicious attacks, ranging from passive eavesdropping to active interference. In wired networks, however, the attacker needs to either access to the physical media e.g. networks wires etc. or pass through a plenty of firewalls and gateways. In wireless networks the scenario is totally different, since there are no firewalls and gateways. So, attacks can be launched by any node in the neighborhood. Every node in the Ad Hoc Network must be prepared to encounter with this adversary.
Each mobile node in an Ad Hoc Network is an autonomous unit that moves freely and independently. This means a node without adequate protection is susceptible to being

captured or compromised. It is difficult to track down a single compromised node in a large network. Attacks stemming from a compromised node are more detrimental and much harder to detect.
Ad Hoc Networks have a decentralized architecture, and many ad hoc network algorithms rely on cooperative participation of the member nodes. Adversaries can exploit this lack of centralized decision making architecture to launch new types of attacks aimed at exploiting these cooperative algorithms. The adversary who compromises an ad hoc node could succeed in bringing down the whole network by disseminating false routing information and this could culminate into nodes feeding data to the compromised node. Hence an intrusion detection system is very important in all types of networks.
The starting point of the development of a secure model should always be the analysis of the attacks. In the following subsection we describe general attacks that could be launched in Ad Hoc Networks.

8.1. General attacks on Adhoc Networks

Attacks on an Ad Hoc Network routing protocol generally are of two categories: *routing disruption attacks* and *resource consumption attack*.
In a routing disruption attack, the attacker attempts to cause legitimate data packets to be routed in a dysfunctional way. In this type of attacks incorrect control messages are injected into the network to subvert or damage routes.
For example: Black hole, misrouting, location disclosure and update storm etc.
In a resource consumption attack, the attacker injects packets into the network in an attempt to consume valuable network resources such as bandwidth, memory (storage) or computation power. Moreover are *Traffic and Data distortion.* This type of attacks can snoop network traffic, manipulate or corrupt packet header or contents, block certain types of traffic or replay transmissions for some malicious purposes.
From an application layer perspective, both attacks are instances of DoS attacks.
In our model we will only concentrate on routing disruption attacks with a focus on source routing, because BeeAdHoc is a source routing protocol.
Particularly in MANETS attacks can be categorized to their consequences and to the techniques which are used. Some of the the consequences are for example:

- Blackhole
- Routing Loop
- Network Partition

- Selfishness
- Sleep Deprivation

There are numerous examples of techniques to archive the consequences described above[aHL03]:

- Cache Poisoning
- Farbricated Route Messages
- Rushing
- Wormhole
- Packet dropping
- Spoofing
- Malicious Flooding

In the following the important attacks are introduced [HPB02] :

Update Storm:
The malicious node deliberately floods the whole network with meaningless route discovery messages or ROUTE REPLY messages. The purpose of this attack is to exhaust the network.

Location disclosure:
Reveals information regarding the location of nodes or the structure of the network.

Replay Attack:
An attacker sends old advertisements to a node causing it to update routing table with stale routes.

Routing loop attack:
This is an example of routing disruption attack. In this case an attacker can send forged routing packets to create a routing loop, which might cause packets to traverse nodes in a cycle without reaching destinations consuming energy and available bandwidth. A loop is inserted into a route path.

Black hole:
An attacker creates a routing black hole, in which all packets are dropped. By sending forged routing packets the attacker could direct all packets for other destinations to itself and then discard them. The attacker could also cause the routes

in all nodes of an area of a network to point into this area when in fact the destination is outside the area. An attacker advertises a zero metric for all destinations causing all nodes around it to route packets toward it. All traffic is redirected to the malicious node which may simply drop all packets.

Gray hole attack:

This attack is a special case of a black hole. An attacker could create a gray hole in which it selectively drops some packets chosen at random or by a certain pattern. For example forwarding routing packets but not data packets, or it might drop a packet each second.

Network Partition:

A connected network is partitioned into k (k > 2) subnetworks where nodes in different subnetworks cannot communicate even though a route between them does exist.

Gratuitous detour attack:

An attacker may also attempt to cause a node to use detours or may attempt to partition the network by injecting forged routing packets to prevent one set of nodes from reaching another. An attacker may tell other nodes that the route through itself is longer by adding virtual nodes to the route.

Wormhole:

A tunnel is created between two nodes that can be utilized to secretly transmit packets. A wormhole in the network is created by a pair of attacker nodes A and B linked via a private network connection. Every packet that A receives is forwarded through the wormhole to B where it is broadcasted by B. Such an attack potentially disrupts routing by short circuiting the normal flow of routing packets and the attacker may also create a virtual vertex cut that he controls.

An attacker records packets at one location in the network, and tunnels them to another location. Routing can be disrupted when only routing control messages are tunnelled.

Cache Poisoning:

Information stored in routing tables is either modified, deleted or injected with false information.

Fabricated Route Messages:

Route messages (route requests, route replies, route errors, etc.) with malicious contents are injected into the network. Specific methods include: **False Source Route:** An incorrect route is advertised in the network, e.g. setting the route length to be 1 regardless where the destination is. **Maximum Sequence:** Modify the sequence field in control messages to the maximal allowed value. Due to implementation issues, a few protocol implementation cannot effectively and finally detect and purge these ”polluted” messages. As a result they can invalidate all legitimate messages with a sequence number falling into normal ranges for a fairly long time.

Rushing attack:[HPB03]

This can be used to refine Fabricated Route Messages. In several routing protocols some route message types have the property that only the message that arrives first is accepted by a recipient.
The attacker simply disseminates a malicious control message quickly to block legitimate messages that arrive later. The rushing attack is a malicious attack that targets against on-demand routing protocols that use duplicate suppression at each node.

Resource consumption attack:

Here an attacker can inject extra data packets into the network which will consume bandwidth resources when forwarded, especially over detours or routing loops. Similarly, an attacker can inject extra control packets in the network which may also consume bandwidth or computational resources as other nodes process and forward such packets.

Packet dropping:[aHFLS02]

A node drops data packets (conditionally or randomly) that are supposed to be forwarded. Packet dropping does not actively change routing behavior as Black hole does. A node launching the attack simply drops data or route packets when it wishes. Based on the frequency and selectiveness, it has the following popular variations: Dropping packets at random, dropping all packets, periodic dropping and selective dropping.
Attackers who do packet dropping can have different motivations. It can simply show selfishness expecting to save power, or intentionally preventing someone else from getting proper service.

Identity impersonation:

Attackers can impersonate another user to achieve various malicious goals. In a networked environment, it is important to correctly attribute a user’s behavior with its proper identity. Pointing to an innocent individual as the culprit can be even worse than not finding the identity of the culprit. In particular, IP and MAC addresses can be used for identification purposes. Unfortunately, these identities are easy to be forged during the transmission of data packets on network or link layers if the underlying communication channel is not encrypted.

Spoofing:
A node injects data or control packets with modified source addresses.

Malicious Flooding:
A node delivers unusually large amount of data or control packets to the whole network or some target nodes.

Selfish Node:
A node is not serving as a relay to other nodes. This way the attacker tries to save its resources. This does no really harm the whole network but reduces the performance.

Sleep Derivation:
A node is forced to exhaust its battery power.

Hop Drop Attack:[HB04]
In this case a forwarding node removes a predecessor node from the source route.

8.2. Cryptographic Routing Algorithms

Most of the traditional secure routing algorithms are based on one of the following routing methods: Ad Hoc on-demand distance vector routing (AODV) and dynamic source routing (DSR). They belong to the class of reactive protocols which only discover routes on demand.
Because of this, both of these algorithms are described in this section, followed by a short overview on types of cryptographic techniques applied to secure them.
Finally, in this section two cryptographic routing algorithms will be described.

8.2.1. AODV

The AODV - Protocol [Ne04] is a reactive protocol which means that every unknown or broken route is discovered on demand. To discover a new route a RREQ (route request) is broadcasted with the address of the source and destination node, a sequence number, a broadcast-ID and the actual hop count. If an intermediate node receives a RREQ it will check whether it is the destination or a valid route to the destination node is in its routing table. If a route is known it will send a RREP (route reply) to the source. If no route is known and it is not the destination then the reversed route is stored and

the packet with an increased hop count is rebroadcasted. If the packet arrives at the destination node then the destination will send a Route Reply message. An intermediate node which receives a RREP stores a forward route to the sender (including its successor on this route) with the corresponding hop count in its routing table. If the RREP arrives at the source node of the RREQ the route is complete and it can be used for sending data.
AODV is able to recognize broken links (e.g. with hello messages). For these cases the protocol uses RERR (route error) messages which are sent to the source node of a route. If more then one RREP arrives at the source node, the best one will be taken.

8.2.2. DSR

DSR [Ne04] is a reactive protocol, too. The discovery of a route is similar to the AODV protocol with the difference that the whole route is stored in the RREQ. If a RREP returns to the source node it is stored in the route cache. For sending data packets on this route a source header is added to the packet and then is sent over the intermediate nodes of the route known.
It is also possible that an intermediate node has a valid current route for the destination of a RREQ. In this case it completes the route for the destination in the RREQ plus the route in the route cache.

8.3. Typs of Cryptography

8.3.1. Asymmetric Cryptography

Protocols like SAODV (as described below) use asymmetric cryptography like RSA to create digital signatures that are used for authentication.
For this kind of cryptography a public key for all nodes and a private key of the node which signed the message is needed.
The following flow shows how a public key mechanism works:

1. Alice does the following:
   - Creates two secret prims: e.g. $p = 13$, $q = 11$
   - Calculates the products $m = p * q = 143$ and $z = (p - 1)(q - 1) = 120$
   - Chooses private key d which is relative prim to z, here $d = 19$
   - Chooses e as $e * d \bmod z = 1$, here e = 139
   - Sets the public key to (m, e)

2. Alice publishes the public key to Bob
3. If Alice sends a message she signs the packet with her private key, e.g. message = 3, 4, 1, 2; signature = 81 (3^{19} mod 143), 69 (4^{19} mod 143), 1 (1^{19} mod 143), 50 (2^{19} mod 143).

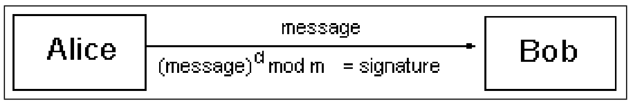


Figure 8.1.: Asymmetric Cryptography: Signing

4. For verifying, Bob calculates $signature^e$ mod n and if this equals to the message, the check is true, e.g. verified message = 3 (81^{139} mod 143), 4 (69^{139} mod 143), 1 (1^{139} mod 143), 2 (50^{139} mod 143).

8.3.2. Symmetric Cryptography

This kind of cryptography works with a common secret key. Examples are the Diffie - Hellman - Protocol which is e.g. used on SSH or the OneTimePads. This kind of cryptography is not often used by cryptographic routing algorithms because is very difficult to distribute the secret keys in the network.
A Protocol introduced by Awerbuch, Holmer and Rubens is ARIADNE that uses this type of cryptography. The technique is quite simple: two parties have the same key for encryption and decryption.
In the Diffie-Hellman protocol parts of the secret key are sent over the public channel, but the hole secret key is only known to the two parties of the protocol.

8.4. Secure Cryptographic Routing Algorithms

8.4.1. SAODV

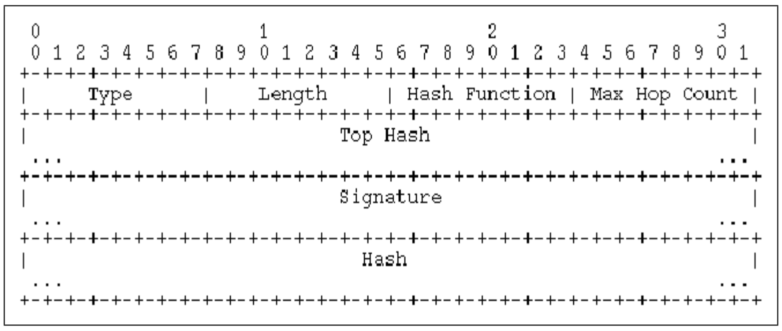
The secure ad hoc on-demand distance vector routing (SAODV) [Pe] [Zap01] [AC03] is based on the AODV - Protocol and adds some extensions to secure AODV.
The extensions include integrity which means that the message will not change along the route and authentication which means that the sender is the same node as indicated in the message.
A metric protection is provided by a hash chain and authentication by a digital signature, and both are added to a special message as described below.
SAODV assumes that there is a central key distribution mechanism so that all nodes

get the certificated public keys of the other nodes in the MANET and that new keys are distributed when a new device joins the Ad Hoc Network.

AODV extensions SAODV has 5 kinds of extensions of the AODV packets:

RREQ (Route request single signature extension)

This is an extension for an easy route request where only the destination nodes are allowed to send a RREP. The non - mutable fields of this extension include the type (here: 64), the length of the extension, a hash function, the maximum hop count, a value derived from the relationship of maximum hop count and hash function and the signature of the extension. The mutable field is the actual hash value defined



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     | Hash Function | Max Hop Count |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Top Hash                           |
 ...                                                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Signature                           |
 ...                                                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Hash                              |
 ...                                                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8.2.: SAODV: Route single signature extension [Zap01]

as a function of the actual hop count.

An intermediate node will store the signatures and the lifetime in its routing table.

RREP (Route reply single signature extension)

This is an extension for an easy route reply which is only sent from a destination node which receives a RREQ. The mutable and non mutable fields equal the fields in a RREQ except the type which is here 65.

RREQ (Route request double signature extension)

A security problem is that in AODV every intermediate node can reply a route if its routing table has a valid entry to the destination node.

This "route caching" is dangerous because the destination node is not involved in the reply of the message and so it cannot sign the message with its signature.

A malicious node could take advantage of this. In order to allow any node to send the route replies, this kind of message adds to the fields of the route request single signature extension and a second signature for a possible route request double signature extension, the prefix size for the RREP and same flags. The type of this message is 66. An intermediate node will store both signatures and the lifetime in its routing table.

RREP (Route request double signature extension)
This extension with type 67 is for a route reply of an intermediate (or destination) node.
It contains the fields of a single signature extension of the signature is from a route request double signature extension of the destination node plus the old life time of the route and a signature of the intermediate node with the actual life time of this route.

RERR (Route error signature extension)
This is an extension of the error messages of AODV and it only includes the signature of the node which finds that its neighbor is unreachable. In AODV it is possible to send "Hello" messages to the neighbor nodes. If a node does not answer the message, there might be an error.

RERR-ACK (Route acknowledgement signature extension)
This is an extension for the acknowledgement messages of AODV and it only includes the signature of the source node.

Route discovering

In the following we show the path of an AODV message with a single signature extension with a focus on SAODV extensions.
The next section explains the function of the hash function and the signature.
At the source node the hash function $\boldsymbol{F}$ is computed as a standard hash function (e.g.MD5). The hash value is $\boldsymbol{F(s)}$ whereas $\boldsymbol{s}$ is a secret random number, the maximum hop value $\boldsymbol{n}$ as the time to live (TTL) -field of an IP packet and finally the hash now becomes $\boldsymbol{F^n(s)}$.
The digital signature contains all fields of the AODV packet plus extensions before the signature field whereas all mutable fields (hop count) are set to "0". The encryption takes place with the private key of the source node. A possible encryption mode is RSA as described above. The RREQ is then broadcasted.
The intermediate nodes take following steps: The node compares the destination IP to check if it is the destination node. If not, it authenticates the signature with the public key of the source node. If the signature comparison returns a true, the node checks whether the hop count is right or not. This checkup is true if the top hash value equals $\boldsymbol{F^{((maxhopcount)-(hopcount))}(hash)}$. The hash value is set to $\boldsymbol{F(hash)}$, the

predecessor node (reversed route) is stored in the routing table with the life time, source and destination of the message and the RREQ is again broadcasted.
If a node is the destination of the packet then it authenticates the signature of the packet withe the public key of the source node and it also verifies the hash value as described for intermediate nodes. The predecessor node is stored in the routing table with the life time and source of the message.
The route reply is sent to the predecessor of the RREQ. It transits the intermediate nodes until it reaches the source.
A source node can also send an RREQ with double signature extension so that intermediate nodes can send an actual route if they have a legal copy of a former RREP for the destination. The RREP is sent with the old signature, the signature from the double signature extension and the former TTL.

8.4.2. ARIADNE

Ariadne is a secure on-demand ad hoc routing protocol which is based on DSR. It prevents attackers or compromised nodes from tampering with uncompromised routes consisting of uncompromised nodes, and also prevents a large number of Denial-of-Service (DoS) attacks.
In addition, Ariadne is efficient, using only highly efficient symmetric cryptographic primitives. Ariadne can authenticate routing messages using one of three schemes:

- shared secrets between each pair of nodes
- shared secrets between communicating nodes combined with broadcast authentication
- digital signatures

Ariadne is most commonly used with TESLA, an efficient broadcast authentication scheme that requires loose time synchronization. Using pairwise shared keys avoids the need for synchronization but at the cost of higher key setup overhead; broadcast authentication such as TESLA also allows some additional protocol optimizations.
TESLA achieves asymmetry from clock synchronization and delayed key disclosure, rather than from computationally expensive one-way trapdoor functions.

9. Artificial Immune Systems

Introduction

The interest in studying the immune system is increasing over the past few years. Computer scientists, engineers, mathematicians and other researchers are particulary interested in the capabilities of this system, whose complexity is comparable to that of the brain. This new field of research is called **Artificial Immune System**.
Many properties of the immune system (IS) are of great interest for computer scientists and engineers:

- uniqueness: each individual possesses its own immune system with its particular vulnerabilities and capabilities;
- recognition of foreigners: the (harmful) molecules that are not native to the body are recognized and eliminated by the immune system;
- anomaly detection: the immune system can detect and react to pathogens that the body has never encountered before;
- distributed detection: the cells of the system are distributed all over the body and are not subject to any centralized control;
- imperfect detection (noise tolerance): an absolute recognition of the pathogens is not required, hence the system is flexible;
- reinforcement learning and memory: the system can 'learn' the structures of pathogens, so that future responses to the same pathogens are faster and stronger.

Without the immune system, death from infection would be inevitable. Its cells and molecules maintain constant surveillance of infecting organisms. They recognize a wide variety of infectious foreign cells and substances, known as non self elements, distinguishing them from native non-infectious cells, known as self molecules.
When a pathogen (infectious foreign element) enters the body, it is detected and the resources are mobilized for its elimination. The system is capable of "remembering" each infection. As a result, the second response to the same pathogen is relatively faster.

The clonal selection principle or theory is the algorithm used by the immune system to describe the basic features of an immune response to an antigenic stimulus.
It provides the idea that only those cells that recognize the antigens proliferate.

9.1. The Human Immune System

The human immune system [dCvZ99] is extremely complex. It has evolved over hundreds of millions of years to respond to invasion by the pathogenic microbes that regularly attempt to infect our bodies, and invasion by the microbes that tried to infect our genetic ancestors. There are similarities between the immune system of humans and those of the primitive of vertebrates, going back five hundred million years on the evolutionary ladder.
The immune system does not rely on one single mechanism to deter invaders, but instead uses many strategies. Some strategies use innate immunity, which does not require previous exposure to the invading microbe. The others use acquired immunity, whereby the immune system 'remembers' how to deal with a microbe that it has dealt with before.
An antigen is any substance that elicits an immune response, from a virus to a sliver. The most important function of the immune system is to be able to differ between self and non self. The other functions are: general/specific, natural/adaptive = innate/acquired, cell-mediated/humoral, active/passive, primary/secondary. Parts of the immune system are antigen-specific (they recognize and act against particular antigens), systemic (not confined to the initial infection site, but work throughout the body), and have memory (recognize and mount an even stronger attack to the same antigen next time).
Self/non-self recognition is achieved by having every cell display a marker based on the major histocompatibility complex (MHC). Any cell not displaying this marker is treated as non-self and is attacked. The process is so effective that undigested proteins are treated as antigens. Sometimes the process breaks down and the immune system attacks self-cells. This is the case of autoimmune diseases like multiple sclerosis, systemic lupus erythematosus, and some forms of arthritis and diabetes.
There are also cases where the immune response to innocuous substances is inappropriate. This is the case of allergies and simple substance that elicits an allergen response.
B-cells are cells, which are mature upon creation by the bone marrow and work through secretion of chemicals called antibodies into surrounding tissue. These antibodies

attach themselves to the antigen, which the B-cell has created to destroy them. The antibodies serve to mark a particular cell for destruction, either through chemical processes evolving proteins or by the cleansing facilities of the liver and spleen.
T-cells work much differently than B-cells; the major differences being that they are created as immature cells and are sent to the thymus to be programmed for the recognition of one specific target antigen. In addition, B-cells are free to circulate in the blood, but they can not penetrate living tissue. In contrast, a T-cell can move anywhere in the body. The final difference is that a T-cell kills those cells that it has been programmed to attack. It does this through direct contact, attaching to the cell and then injecting strong chemicals into the target cell.
T-cells serve other functions as well. They can assist B-cells by triggering a chain reaction which causes the B-cell to form plasma cells. These plasma cells then begin to produce more B-cells thus allowing the number of antibodies to increase exponentially. Other T-cells then have the job of monitoring the level of antigen and stopping the production of B-cells when the infection has been destroyed.

9.2. How the Immune System protects the body

Our body is protected by a diverse army of cells and molecules that work together to detect an antigen (Ag), which is usually a foreign molecule from a bacterium or other invader. Figure 9.2 shows a simplified version of the basic immune mechanisms of defense. Specialized antigen presenting cells (APCs), such as macrophages, roam in the body, ingesting and digesting the antigens they find and fragmenting them into antigenic peptides (shown as I in figure 9.2). Pieces of the peptides are joined from histocompatiblity complex (MHC) molecules and are displayed on the surface of the cell.
Other white blood cells, called T cells or T lymphocytes have receptor molecules that enable each of them to recognize a different peptide-MHC combination (shown as II in figure 9.2). T cells activated by that recognition divide and secret lymphokines, or chemical signals, that mobilize other components of the immune system (shown as III in figure 9.2). The B lymphocytes which also have receptor molecules of a single specification on their surface, respond to those signals. Unlike the receptors of T cells, however, those B cells can recognize parts of antigens that are free in solution, without MHC molecules (shown as IV in figure 9.2). When activated, the B cells divide and differentiate into plasma cells that secrete antibody proteins, which are forms of their receptors (shown as V in figure 9.2). By binding to the antigens they find, antibodies can neutralize them (shown as VI in figure 9.2) or precipitate their destruction by complement enzymes or

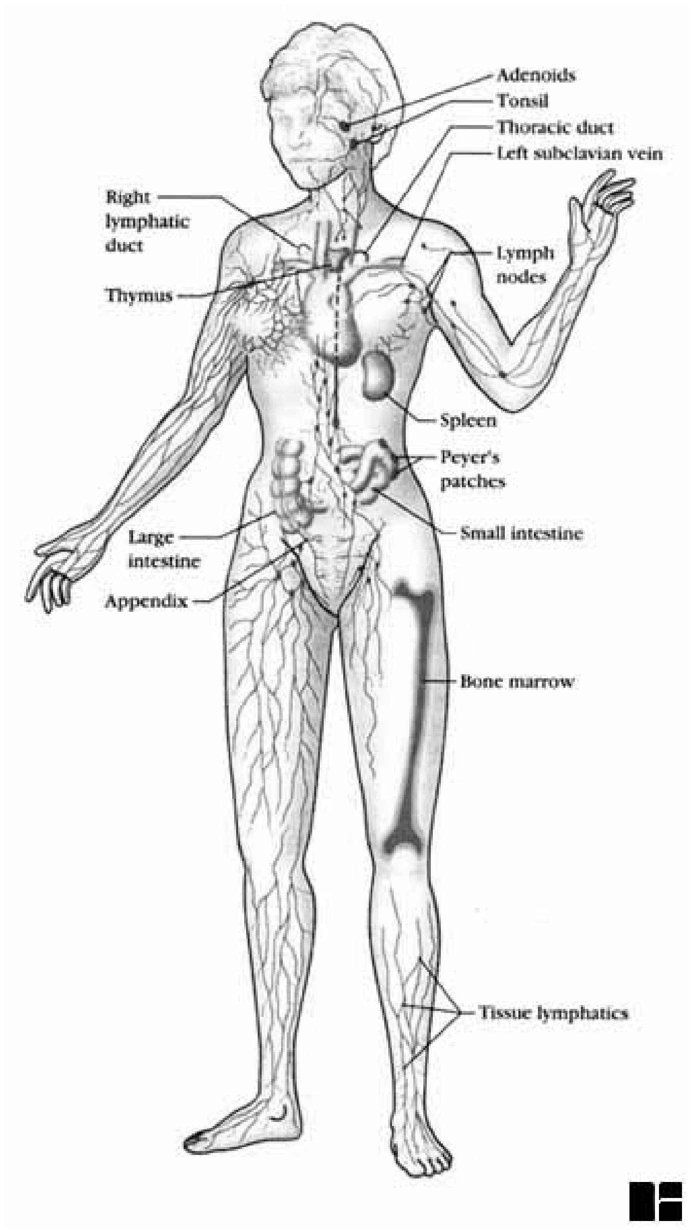


Figure 9.1.: The Anatomy of the Human Immune System[Imm]

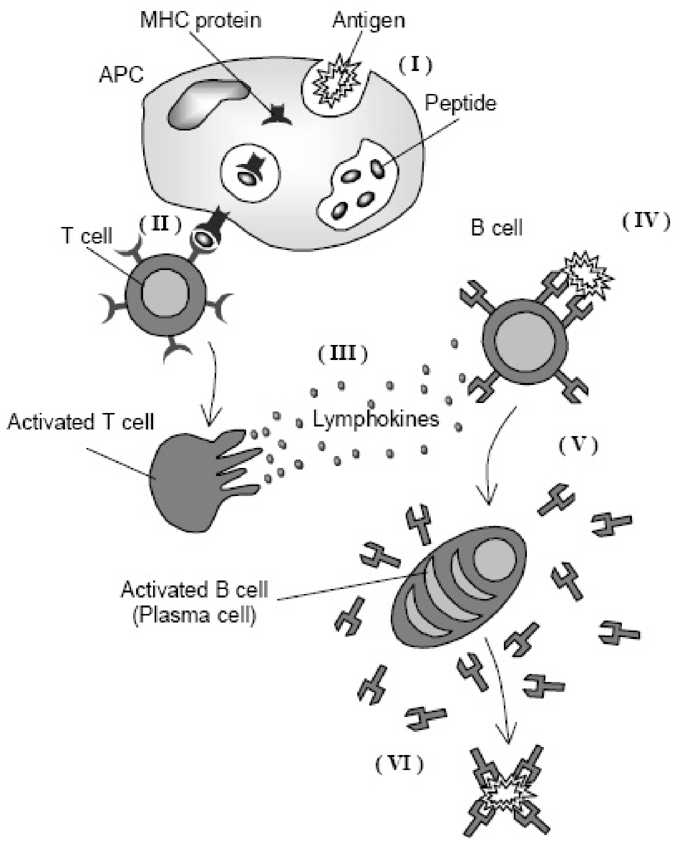


Figure 9.2.: How the immune system defends the body [dCvZ99]

by scavenging cells.
Some T and B cells become memory cells that persist in the circulation and boost the immune systems's readiness to eliminate the same antigen if it presents itself in the future.[ES03]

9.2.1. The Immune Cells

The immune system is composed of a great variety of cells that are originated in the bone marrow, where plenty of them mature. From the bone marrow, they migrate to patrolling tissues, circulating in the blood and lymphatic vessels. Some of them are responsible for the general defense, whereas others are "trained" to combat highly specific pathogens.
For an efficient functioning, a continuous cooperation among the agents (Cells) is nec-

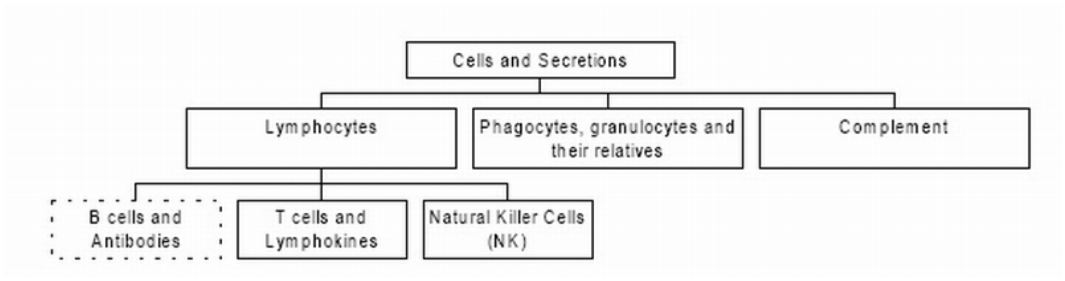


Figure 9.3.: Structural division of the cells and secretions of the immune systems [dCvZ99]

essary. Figure 9.3 presents a structural division among the cells and secretions that are produced by the immune system.

In the following subsections we will present in detail how the immune system manage the pattern recognition, reinforcement learning and the discrimination between self and non-self.

9.2.2. Reinforcement Learning and Immune Memory

In order to be protective, antigen recognition is not enough. The immune system must also provide resources to effectively combat pathogens.
As in typical predator-prey situations, the size of the lymphocyte subpopulation which is specific for a pathogen (clone) is crucial for determining the outcome of infection.
Learning in the immune system involves raising the population size and affinity of those lymphocytes that have proven themselves to be valuable by having recognized some antigen. Because the total number of lymphocytes in the immune system is regulated, therefore increase in size of some clones is to be complemented by a corresponding decrease in amount of other clones.
However, the total number of lymphocytes is not kept absolutely constant but if the immune system would learn only by increasing the population size of specific lymphocytes, it had either to drop some previously learned antigens, increase in size or had to constantly decrease the portion of its repertoire that is generated at random and responsible for responding to novel antigens.
In the normal course of the evolution of the immune system, an organism would be expected to encounter a given antigen repeatedly during its life time. The initial exposure to an antigen that stimulates an adaptive immune response (an immunogen) is handled

by a variety of clones of B cells, each producing antibodies of different affinity.
The efficiency of the immune response to secondary encounters could be considerably enhanced by storing some high affinity antibody producing cells from the first infection (memory cells), so as to form a large initial clone for subsequent encounters. Rather than 'starting from scratch' every time, such a strategy would ensure that both the speed and accuracy of the immune response becomes successfully greater after each infection.
This scheme represents a a reinforcement learning strategy, where the system is continuously improving its capability to perform its task.[dCvZ99]

9.2.3. Pattern Recognition

From the point of view of pattern recognition in the immune system, the most important feature of both B and T cells is that they have receptor molecules on their surfaces that can recognize antigens (either free or bound to an MHC molecule). In the B cells case, the receptor is an immunoglobin or antibody molecule embedded in the membrane of the cell. In the T cells case, the receptor is simply called the T cell receptor (TCR). Complementarity recognition in the immune system occurs at the molecular level and is based on shape between the binding site of the receptor an a portion of the antigen called an epitope. While antibodies possess a single kind of receptor, antigens may have multiple epitopes, meaning that a single antigen can be recognized by different antibody molecules. B and T cell receptors recognize different features of an antigen. The B cell receptor interacts with epitopes that are present on intact antigen molecules. Antigen molecules may be soluble or bound to a surface. The T cell receptor interacts only with cell surface molecules. T cells secrete chemical substances that can kill other cell or promote their growth, playing a very important role in the regulation of the immune responses. By recognizing a cell surface molecule, the T cell has to detect whether it is interacting with another cell or with a soluble molecule. The T cell receptor recognizes antigens bound to a cell surface molecule called a major histocompatibility complex (MHC) [dCvZ99].

9.2.4. Immunologic Self/Nonself Discrimination

It is often said that the repertoire of cells in its capability to recognize pathogens is complete and there is a large stochastic nature of antigen-receptor formation. This represents a fundamental paradox,because all molecules (shapes) can be recognized including our own cells, which are also seen as antigens or self-antigens.
For the immune system to function properly, it needs to be able to distinguish between

the molecules of our own cells (self) and foreign molecules (non self) which are a priori indistinguishable.
If the immune system is not capable of performing this distinction, then an immune response will be triggered against the self-antigens,causing autoimmune diseases. Not responding against a self-antigen is a phenomenon called self-tolerance or simply tolerance. Understanding how this is accomplished by the immune system is called the self/nonself discrimination problem [dCvZ99].

9.2.5. AIS Definitions

There are three definitions for an Artificial Immune System:

- The AIS are data manipulation, classification, representation and reasoning strategies that follow immune system [dC].
- The AIS are composed of intelligent methodologies, inspired by the biological immune system, toward real-world problem solving [Das98].
- An AIS is a computational system based on natural immune system metaphors [Tim].

The following directions are shaping AIS research:

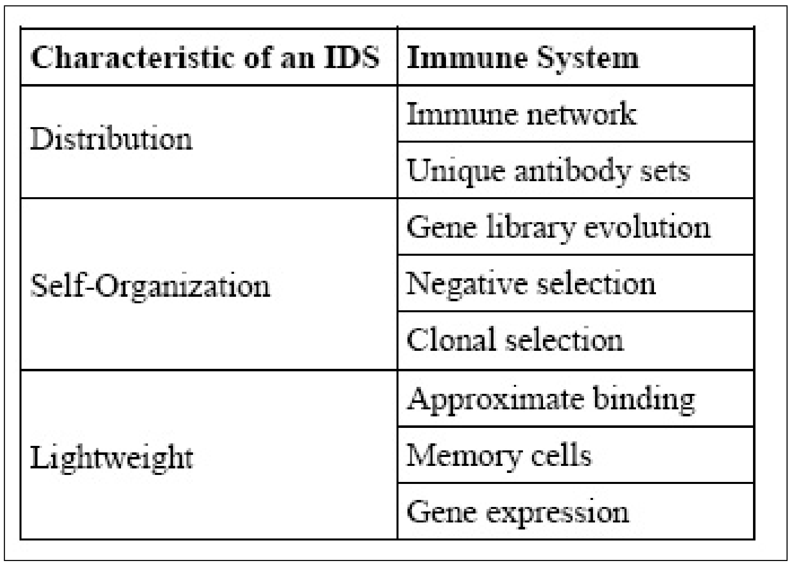
- Hybrid structures algorithms that take into account immune-like mechanisms;
- Computational algorithms based on immunological principles, like distributed processing, clonal selection algorithms, and immune network theory;
- Immunity - based optimization, learning, self organization, artificial life, cognitive models, multi agent systems, design and scheduling, pattern recognition, anomaly detection, computer and network security and
- Immune engineering tools

9.3. IDS

Now one can try to see the analogy between the human immune system and network intrusion detection system.
There are three fundamental requirements for the derivation of the design goals for network-based intrusion detection namely: *distribution, self-organization and being lightweight*. Fig. 9.4 describes the immune system components and functions that satisfy the three listed requirements [dCvZ00].

9.3.1. Inspired by Nature

In nature the immune system protects a body. These natural immune systems can distinguish between cells of the body (called self) and intruders in the body (called non-self). Because it is very efficient, we could possible use it to provide security in a network with small computational and communication complexity.
An Artificial Immune System (AIS) can be used to distinguish normal activities and abnormal activities. If we can distinguish between good and bad nodes, we can find a route through a network where there are only trusted nodes. The behavior of such a system is very flexible, so they that it can counter new attacks.

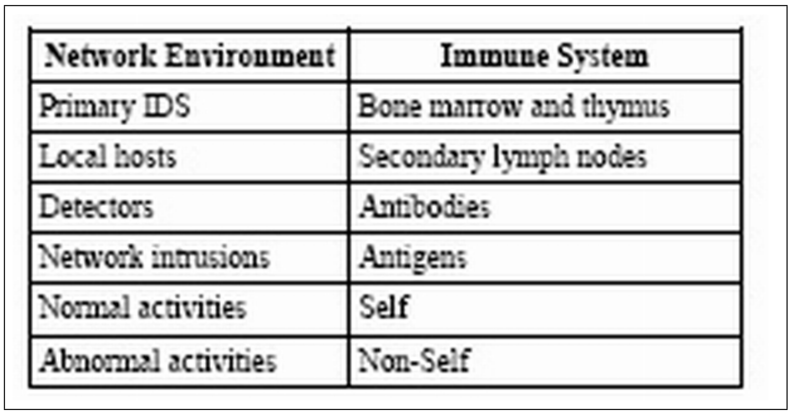


Characteristic of an IDS	Immune System
Distribution	Immune network
	Unique antibody sets
Self-Organization	Gene library evolution
	Negative selection
	Clonal selection
Lightweight	Approximate binding
	Memory cells
	Gene expression

Figure 9.4.: Components of the immune system that satisfy the requirements for the development of an efficient intrusion detection system (IDS) [dCvZ00]

9.3.2. AIS Framework

Since the BeeAdhoc routing protocol is inspired by honey bee behavior, we are convinced that AIS is a good solution for its security.
AIS is in principle also inspired by Nature, because AIS imitates the functionality of the human immune system. The human immune system is very complex in its structure and functionality, but there are some interested aspects for computer engineering. In our case it is the security of ad hoc networks with focus on routing.



Network Environment	Immune System
Primary IDS	Bone marrow and thymus
Local hosts	Secondary lymph nodes
Detectors	Antibodies
Network intrusions	Antigens
Normal activities	Self
Abnormal activities	Non-Self

Figure 9.5.: Relationship between the network intrusion detection systems and the immune system[dCvZ00]

To develop an AIS for thus a problem following steps are required.[Dil]

1. Definition of the problem that we want to solve.
2. Selection of applicable immune principles.
3. Construction of the AIS.
4. Interpretation of the created AIS.
5. Testing of the created AIS.

First we have to define the problem. This step have been already done: to create a secure BeeAdHoc routing algorithm.
The next step is to choose some applicable immune principles. For example:
Pattern recognition: On the basis of some special pattern our system should be able to detect the intruders.
Anomaly detection: The body can detect and react to new pathogens. Similarly, our system will be able to recognize malicious nodes or invaders.
Reinforcement learning and memory: The immune system 'learn' the structure of pathogens, so that future responses to the same pathogens are faster. We have to make sure, that our system should learn and react faster to invaders.
Immunologic Self/Nonself Discrimination: For the immune system to function properly, it needs to be able to distinguish between molecules of our own cells (self) and foreign molecules (non self) which are a priori indistinguishable. Simultaneous our system should also be able to distinguish between self and non self nodes.
Affinity maturation This property describes the fixation between cells. In our model one can compare it as the relationship between a node and its neighbor.

10. Measuring Energy Consumption

Introduction

Since the primary object of our security model is energy consumption, therefore we need a mechanism for measuring this metric in order to make statements regarding the performance of our security extension.
Tracking energy consumption in a computer is a complex task and in the following we present few approaches how to handle this problem and finally describe our solution. After describing the implementation of our profiling framework in NS-2 [ns2] we will present the results of tests we did with different routing protocols.

10.1. Profiling

10.1.1. Energy Model

Tracking the energy consumption, caused by software running in a computer is not a trivial task because there are different locations at which energy is consumed.
They can be categorized as following:

- **The Processor and Memory**
  Every operation in a program that is executed consumes the processor's energy.
  A program also requires access to the memory and so every read and write operation in memory consumes additional energy.
- **The Network Device**
  Another device of interest is the network device, which consumes power during the sending and receiving of packets.
  Since the amount of traffic depends on the routing protocol, therefore the network device consumes a significant amount of energy.
- **The Battery**
  The battery provides the energy to the above mentioned devices.

There have been a lot of approaches published how to calculate the energy consumption. The main problem is to get accurate results without any error that is generated by the

environment in which a programm is executed in, including the OS.
So getting accurate values requires an additional hardware like a multimeter. Software based solutions are less complex but they then return less accurate results.
Since we didn't have the resources for such an evaluation, we concentrated on software based solutions.
There are different approaches that describe software based tracking of energy consumption.
For example:

- Cycle Counts [RJ98]
  This method is a so called proxy metric technique. The technique of proxy metrics is to calculate an unknown measure by known ones.
  One method is to count cycles during the execution of a program and then multiply them with a certain constant that describes the energy consumed during a cycle.
- Operation Tracking [TMW94]
  Another solution is to attach a cost constant to every machine level instruction after deriving different cost constants for every instruction with the help of a multimeter. After that, the sum of the operations used in the code is calculated by considering the differences due to individual sequences in which the operations are arranged. Taking into account the different sequences assembler operations is important because the order of operations changes the energy costs.
- Event Counter [JM01]
  This technique works in a similar way as operation tracking but instead of analyzing the assembler instructions, the system is observed and whenever events take place energy constants for these events are summed up.

Finally we decided to use the cycle count approach to figure out the energy consumption because we were interested in tendencies of energy behavior of different algorithms rather than accurate values.

10.1.2. Profiling Framework

Our profiling framework that gives us the cycle counts consists of two parts:

- Part I : Cycle Counts
  As already mentioned an important source of power consumption is the processor. Among the different approaches we count processor cycles during the execution for programms.

For counting those cycles, we use a small C++ class that provides methods to calculate the cycles executed between two points in an execution.
Some parts of this class are written in assembler to return accurate values. Our class was initially developed for windows and we migrated it to Linux. We replaced the assembler code with a Linux macro. The results from both frameworks were similar, therefore we took the macro for the sake of simplicity.
This class provides several public methods listed in the following overview:

- void startProfile()
  This method initializes the profiling procedure and stores the current cycle count by calling rdtsc.
- signed int64 endProfile()
  In this method the final cycle count, after the execution of the segment that has to be tested, is stored and the number of cycles executed during the monitored time is calculated and returned.
- signed int64 estimatedFrequency()
  In order to calculate the time needed for a cycle this function returns the amount of cycles passing during 1 second of sleep.
- unsigned int findBase()
  This method calculates the overhead of using two assembly instructions cpuid and rdtsc. Finally, it subtracts them from the cycle count. This function also ensures that the instructions are already in cache when the profiling begins so that no cache effects influence the monitoring results.

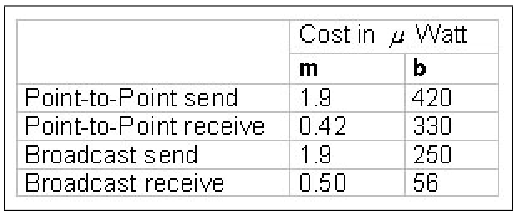
We didn't consider the memory operation although it would be necessary for more precise results.
Since the cycle count mechanism is more precise for short sequences of code we used a parser described in the next part to track the cycle counts of certain parts of the protocols in addition to counting for complete simulation runs.

• Part II : Traffic Tracking
During our simulations of the routing protocols we use a simulator that generates a log file for each run.
The simulator also provides a parser, written in perl to analyze the log files.
By using a slightly modified version of this parser we get the needed information about the traffic generated by a protocol from which we could calculate the energy consumption for the network device.

We take the information about the amount of packets and their size provided by the parser and use fixed constants on these values to get the values we needed.
For the interpretation of this information we use a model based on an approach by Laura Marie Feeney from the Swedish Institute of Computer Science [Fee99].
In her model Feeney provides a basic equation:

cost = m * size + b

This equation describes how to calculate the energy costs of a single packet that is send through the network. m is a constant that is multiplied with the size of a packet and describes the interrelationship between packet size and energy cost. b is another constant that represents the basic energy cost that is consumed at a network device as an activation cost for the sending/receiving mechanism.
Based on this equation, in this model traffic costs are calculated for sending and receiving. So the constants for calculating the energy costs of packets that are used in the formula depend on if a packet has been sent or received.
Feeney also makes a difference between broadcast and p2p traffic and so we use the constants she described in this paper, to calculate the different ratio of different traffic types:



	Cost in µ Watt	
	m	**b**
Point-to-Point send	1.9	420
Point-to-Point receive	0.42	330
Broadcast send	1.9	250
Broadcast receive	0.50	56

- Part II : The Battery
  For evaluating the behavior of the battery we rely on the in NS-2 integrated battery model.

In the beginning we sum up the results from the cycle counts multiplied with a constant for average power consumption per cycle and the results from the network traffic measurement.

10.2. NS-2

Since we don't have the equipment for testing our network routing protocols in a real environment we use the network simulator NS-2. The advantages of using this simulator are that there is already an implementation of BeeAdHoc and some other protocols are implemented for for NS-2.

10.2.1. NS-2 Modifications

To measure the processor cycles during the simulation of a whole scenario, some changes inside the simulator.cc are needed that make calls to our profiling routines:

```
static class SimulatorClass : public TclClass {
    public:
    cTimer ctimer;

    SimulatorClass() : TclClass("Simulator") {}

    TclObject* create(int argc, const char*const* argv) {

        printf("start profiling");
        ctimer.startProfile();

        return (new Simulator);
    }

    ~SimulatorClass() {
        unsigned long long result = ctimer.endProfile();

        printf("end profiling");
        printf("%li cycles have been consumed during execution.",
            result);
    }
};
```

Listing 10.1: Profiling in NS-2 Methods

As shown in the code above, the first measurement is taken in the constructor of the simulator class and the final one in the destructor.
The cycle amount from a whole simulation gives a tendency about the processor com-

plexity. In addition we extended the log part of NS-2 such that the cycles used during handling of network packets can be measured. This information is added into the trace file output and later extracted by the parser from those files to calculate the shares of energy needed by data and control packets.

10.2.2. Testing Results

Introduction

To test the profiling framework and gather more detailed information about the energy consumption we set up same scenarios for the routing protocols AODV, BeeAdHoc and DSR in the NS-2 simulation environment.
During the creation of an earlier version of BeeAdHoc already tests where conducted on the energy consumption of those protocols, but they only focused on the network traffic.
Our goal is now to get a complete picture that also includes the processor energy consumption.

Testing Scenario

For the different simulation runs we used a .tcl file to set an automatically generated testing environment containing the following fixed key parameters:

- Simulation Time : 400 seconds
- Area Size : 500 x 500 meters
- Movement Speed : 5-15 meters per second
- Packets per Second : 0.05
- Paketsize : 512 byte

For two parameters we used variable settings:

- Nodes : 10, 30 and 50
- Pause Time : 1, 30 and 60 seconds

For all simulation runs we set TCP as the transport protocol.
Since we need to calculate average values for the cycle measurement we run each scenario four times. Finally, we did nine combinations of parameters for each protocol and four runs for each combination we got 108 simulation runs for all protocols.

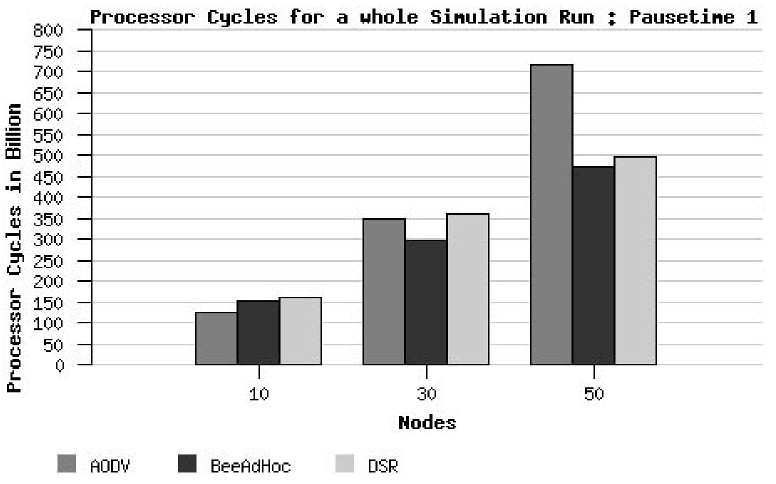
The first measurement shows the amount of cycles consumed during a complete simulation run of the different protocols run with the scenario described above. Each protocol has been tested four times and then an average value is calculated.
In each of those four test runs, all three protocol have been tested rotational to get better comparable results than we would get when we would have tested every protocol stand-alone.
During those tests we received results with an acceptable variance. See figure 10.1.

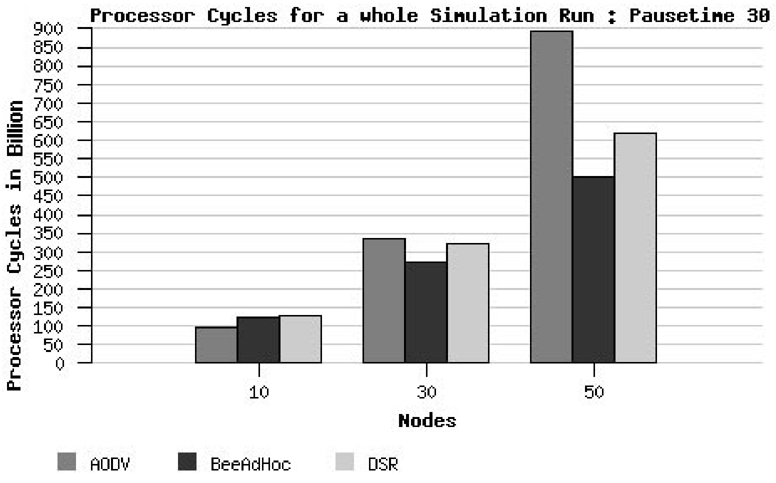
The results for the network traffic energy consumption had been taken from the parser files that have been generated for each simulation run. These values didn't change during the repetition of identical setting. Figure: 10.2

Figures 10.3 - 10.5 represent the amount of cycles consumed for processing either data or control traffic. Those values are also taken from the parser file which collected those values logged by the NS-2 trace function.

Conclusion

Examining the result for the complete cycle test, it becomes obvious that the test conducted on ten nodes doesn't show any significant difference. All three protocols deliver quite close results for this network.
But once we increased the number of nodes we received a picture that kept nearly constant during each change of pause time. BeeAdHoc showed the best energy performance in each test since it needed least number of cycles during the different test runs. DSR always stayed behind BeeAdHoc being a bit more energy inefficient. AODV delivered the worst result for node numbers above ten and it becomes obvious that the larger the network get the worse AODV could handle the situation.
According to the network traffic energy consumption BeeAdHoc started on the second place needing a bit more energy than DSR, except the run with pause time one and network nodes 10 where it showed the worst performance of all three protocols. But with an increasing pause time BeeAdHoc showed the best results for the rest of the simulation runs, as compared with DSR and AODV. AODV had the worst results in eight out of nine simulation scenarios dropping further behind with an increasing pause time.
Regarding to the ratio of the data/control share of consumed cycles, BeeAdHoc gave optimum results. Even in the worst scenario it never increased above 18 %. Meanwhile





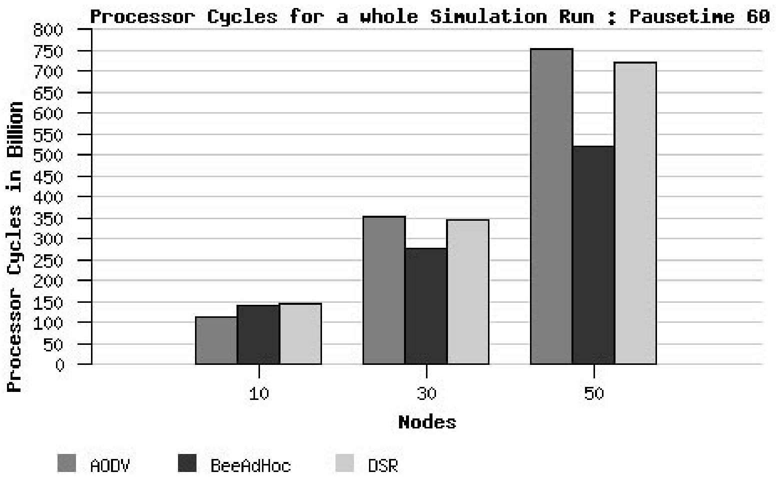
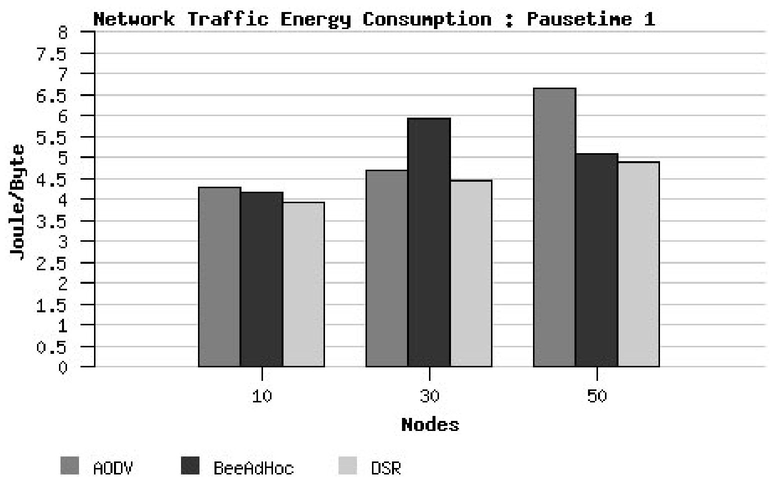
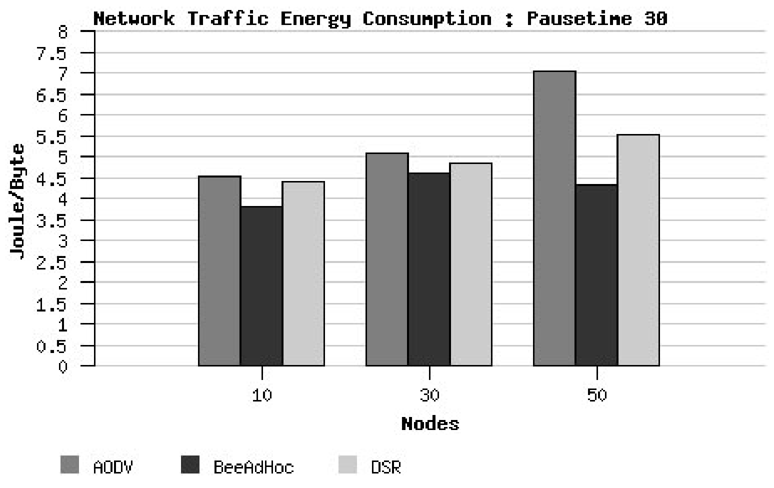


Figure 10.1.: Complete Cyclecounts





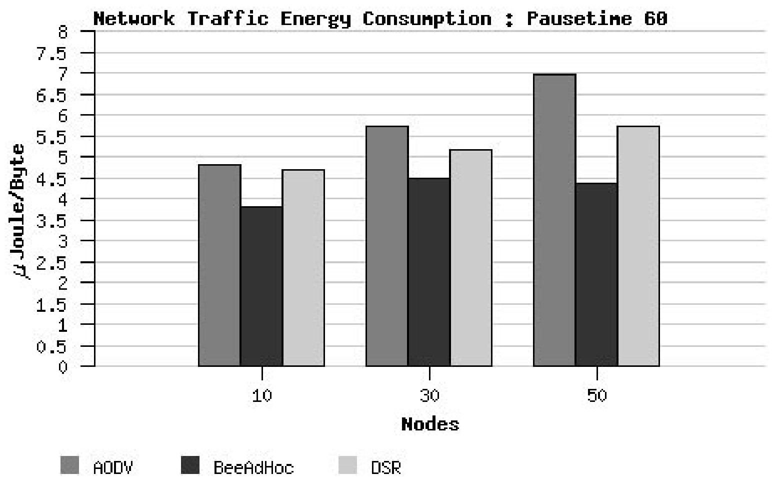


Figure 10.2.: Network Traffic Energy Consumption

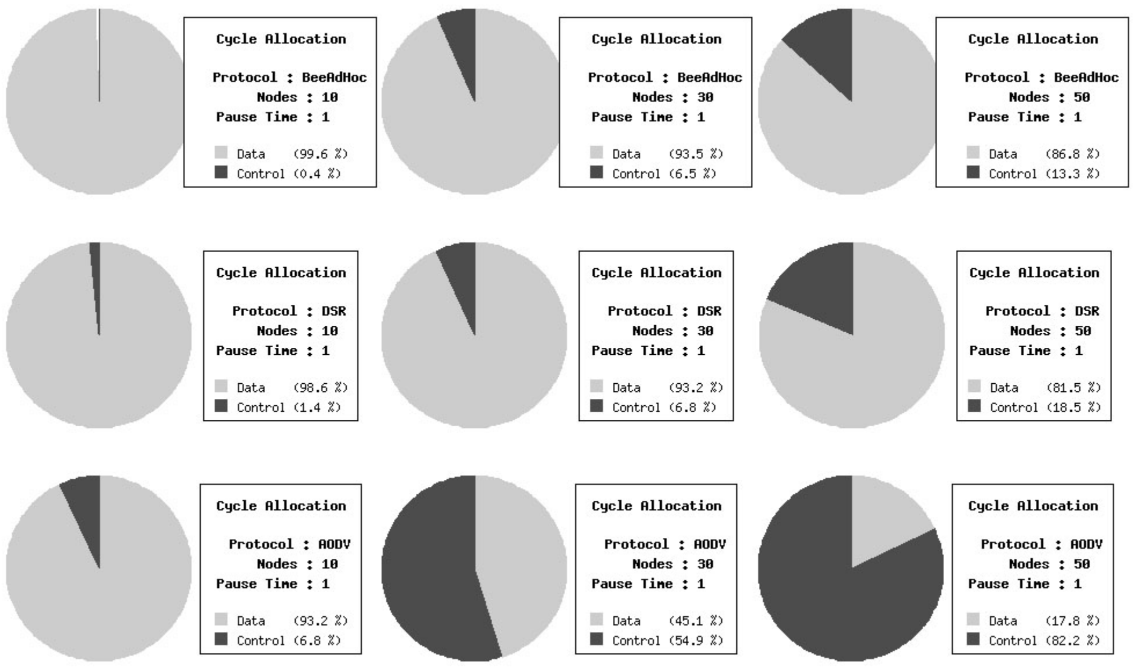


Figure 10.3.: Cycle Allocation: Pausetime 1

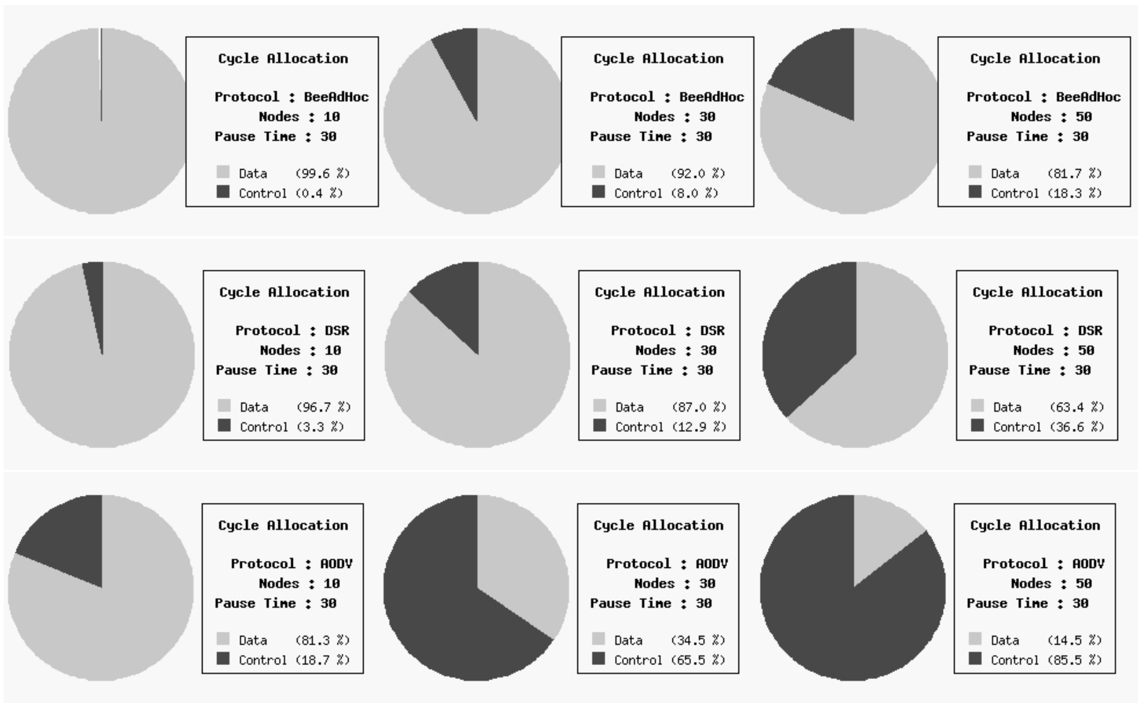


Figure 10.4.: Cycle Allocation: Pausetime 30

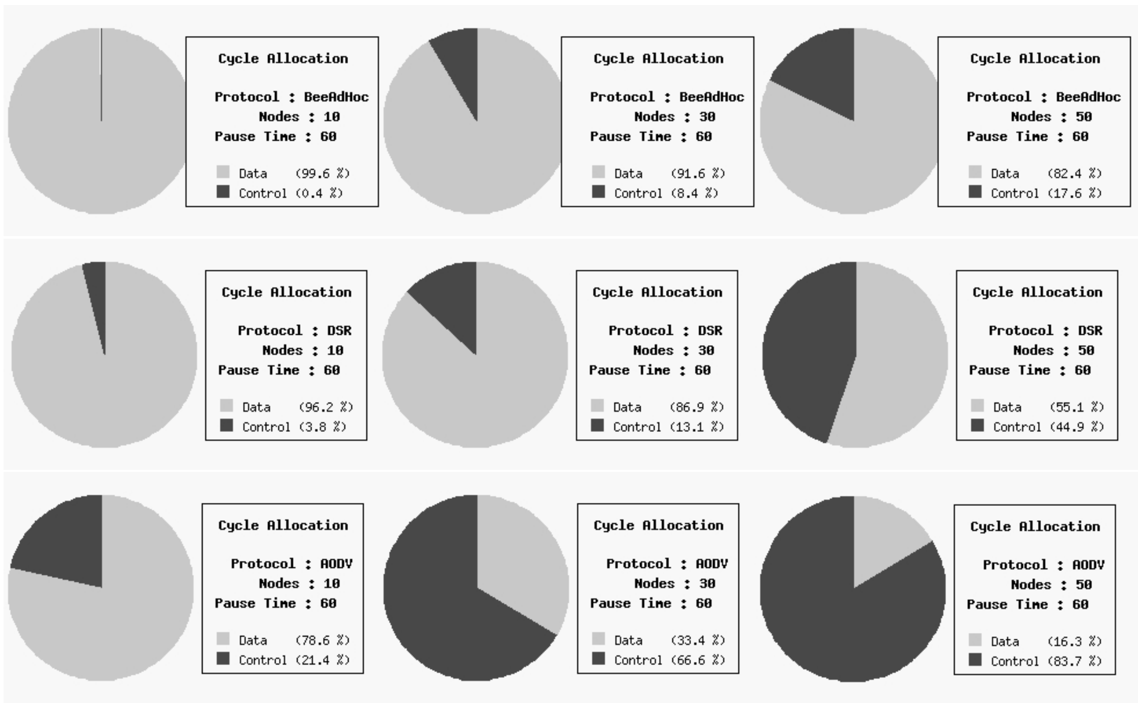


Figure 10.5.: Cycle Allocation: Pausetime 60

DSR reaches a maximum value of about 45 % and AODV showed the worst performance with a value beyond 83 %.
Summarizing the results, it is evident that BeeAdHoc has the best energy performance among three protocols tested. According to the energy consumption for handling the network traffic as well as the consumption of processor energy.
The second place in a overall consideration goes to DSR which is most of the time located between AODV and BeeAdHoc. The worst results came from AODV, especially on large networks.

11. BeeGuard

11.1. Security Model

BeeGuard is the result of our efforts to design a security model for BeeAdHoc that consumes little energy and that fits into the nature inspired background of BeeAdHoc. After the theoretical description of the model, the explanation of the concrete implementation inside NS-2 will follow.
Before we go into detail we want to clarify, the use of a defensive concept. That means if in our model a node detects a bad node it just sets up defensive actions. Another way to deal with such a detection could be to try to neutralize the dangerous node or to inform neighbor nodes about the presence of such an attacker. We discussed quite a long time about the several ways of handling bad nodes but came to the conclusion that every offensive mechanism against bad nodes could also be used by an attacker against good nodes. For example if we include a mechanism for informing other nodes if an intruder is detected the attacker could use this to denunciate good nodes as attackers. After a short time every node believes his whole neighborhood is evil. So we limit our counter mechanisms to defensive ones that just detect the attacks and prevents them. Another important point is, that we tried to avoid using encryption in our security model.
During our research in the beginning we collected a lot of information about different encryption techniques and came to the conclusion that for a mobile network only asymmetric encryption would be an option. The other might be an hybrid mode that

has an encryption part. But this technique has a great disadvantage that is counterproductive to our goal of developing an energy efficient security extension. The following quote from taken from a knowledge base [sil] describes this difficulty:

"Asymmetric encryption
...
The most well-known type of this class of algorithm is the RSA procedure (named after its three inventors). The RSA algorithm is based on the observation that it is quite simple to multiply two very large numbers; the reverse, however, i.e. the factorization of the product into just these two numbers, is an extremely difficult problem.
The disadvantage of this procedure as opposed to symmetric encryption is that it takes a great deal more computing power.
..."

Unfortunately we have no concrete values to show the energy consumption for asymmetric encryption on the system we used for developing and testing BeeGuard. But from dealing with a concrete implementation and from various information sources like the one quoted above we are sure that with such a technique our energy results would be not as good as they are in the current implementation where we have been using alternative mechanisms.

In the beginning we divided up all possible threats in networks: The first group consists of attacks we don't have to consider since they are not relevant for us because of reasons we will describe in the following. The other group consists of attacks that we do have to consider because they could disturb our network and will also introduce our counter mechanisms for them.

11.1.1. Examples of Non Relevant Attacks for BeeAdHoc

- Attacks on Routing Tables
  BeeAdHoc is a source routing protocol. So no routing tables are used to store path information and all attacks on those tables can be ignored for BeeAdHoc. All attacks that try to manipulate the routing tables are irrelevant for us.
- Black Holes
  Black holes describe nodes that destroy every incoming packet. This is no threat for BeeAdHoc because if a forager doesn't return from a destination it couldn't dance and so no route over this bad node will be established.
  This way all similar kind of attacks wouldn't work on BeeAdHoc.

11.1.2. Relevant Attacks for BeeAdHoc

- Flooding
  To handle this attack scenario we developed a technique called antibody.
  Antibodies are created whenever a foreign scout enters the hive. One antibody is created for each source node that enters the hive. Those antibodies have a special property named immunity that is increased each time such a contact takes places and is decreased slowly by regular intervals.
  Whenever this value drops below zero that entry is removed from the node since there is obviously no current threat anymore. If a foreign scout enters a hive and a matching antibody with a certain level of immunity is found, the scout is destroyed. This way nodes that produce an unusual number of route requests are automatic blocked after some time.

- Rush
  A rush is a special kind of DoS attack mechanism. Instead of flooding the network the task of a rush is to reach a node with a faked route request before a real one arrives so that the real one is blocked. In BeeAdHoc duplicates of these scouts that have been seen already at a node are dropped.
  This blocking mechanism should prevent processing of duplicate requests but also enable these rush threat. The aim of a rush attack is that the malicious node can establish a route including itself and prevent other routes that does not contain it. To prevent such an attack every time a scout arrives at a node where a copy of it has already been seen and the missing part of it's route is not available at the node's dance floor, the scout is frozen.
  In case that the unknown part of it's route is already available because the current node has the missing information, the route is completed with this information and the scout returns.
  If the missing part of the route is not available, the bee is kept frozen until either a certain duration has passed or a scout returns to that node carrying the desired information. So every time a scout returns from to it's destination the freeze list of that node is checked for waiting scouts and if there are any they are provided with the missing information. Of course multiple versions of the same scout already on the freeze list are not freezed again but deleted.
- Attacks on the Packet Information
  Since we focus on a source routing algorithm it's very important to secure the rout-

ing information attached to each packet as well as the data carried by it.
The goal is to make it impossible that other nodes can modify the route or the data content of a packet once it has been launched into the network without that this modification is discovered.
The best way to achieve this is asynchronous encryption. But the problem with this encryption method is it's high time and energy consumption and since low energy consumption is a constraint for our work we had to provide another approach.
Whenever a node releases a forager into the network, a pattern for that forger's content inside the source node is updated. Also a checksum for an another route with the same destination is attached to that forager. That route is chosen by random but cannot be the same as the one the forager itself will take.
Checksums represent the state of a pattern. Always the current checksum is attached to a forager. Older entries are stored in a checksum history in the node that contains the pattern.
When that forager arrives at it's destination a mirror pattern is updated, too. If the forager is carrying a valid checksum the pattern list of that node is examined for an existing entry for that route. If such a pattern entry is found the history of that entry is examined, if there is a matching checksum. If not, the connection is insecure and set onto a black list that blocks further attempts to use that route. If a matching entry in the checksum history is found, all older entries are deleted.
Those traffic pattern are either created by scouts that reach their destination or return to their source during a route request or later by incoming foragers if no pattern is available yet.
This mechanism takes advantage of the fact that BeeAdHoc allows multiple routes between a pair of source and destination, so that checksums can be passed on different routes. So a malicious node can fake routing information, destroy packets or create fake packets for another node but since the checksums are traded on other routes it would be very difficult for a bad node to suppress the detection of his attacks.
For this mechanism we suppose that there are always at least two routes available for a pair of source and destination. If this is not the case the mechanism wouldn't work.
But since BeeAdHoc is a protocol for a mobile network we assume that in a decent time frame there will be always different routes between two nodes due to mobility and appearing of new participants. In a static network with dead end nodes this approach wouldn't work.

- Grey Holes
  In difference to a black hole the grey hole just kills a few packets and are therefor is very hard to detect. Since in that kind of an attack some forager will survive and therefore reach their destination and keep this connection alive. This case is also handled by the pattern recognition. The missing packets will result in a false pattern. In this way a malicious node is detected.
- Selfish Nodes
  Selfish nodes deny routing requests to avoid spending their energy on processing foreign requests. In a network every participant has to contribute and so selfish nodes aren't a real danger but annoying, since they decrease the overall network lifetime.
  It's not very easy to detect such parasites so instead of spending a lot of resources to find them we use a quite simple reactive technique.
  For each neighbor node of a hive a so called Honey Pot is created in the moment the hive deals with one of its neighbors for the first time. Whenever the hive needs a certain neighbor to forward an own request the pot is filled with a bit of honey. Each time the neighbor asks the hive to process one forager this forager consumes a bit of honey from the pot. Also the pot is refilled from time to time with a small rate of honey.
  The less honey is in the pot the less or rather the slower request from a neighbor are processed. So after some time the traffic between the hive and a selfish node will be very limited. Of course this mechanism must be balanced for making sure that a node at a dead end of the network isn't treated as selfish.
  It seems to be illogical that only direct neighbors are rated when they pass a packet to the current node, even if they are not the source of the forager. The reason for this approach is that there is no guarantee that the sender stored in the packet is really the true creator of it, since we have no authentication mechanism.

11.2. NS-2 Implementation of BeeGuard

To test the behavior in different attack scenarios and of course to profile the energy consumption of BeeGuard it has been implemented into an existing version of BeeAdHoc inside the NS-2 simulation environment.
In the following the different parts of BeeGuard and the way it has been embedded into BeeAdHoc will be introduced and explained in detail.

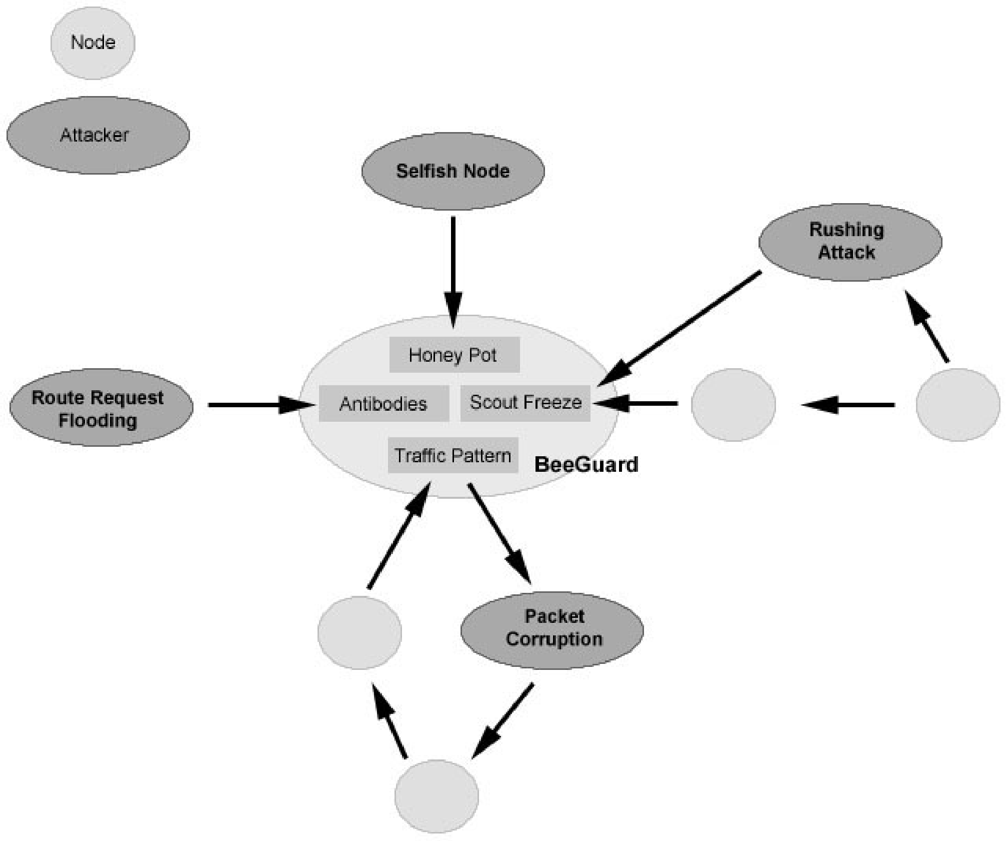


Figure 11.1.: BeeGuard : Theoretical Model

11.2.1. BeeGuard::AntiBody

The task of the antibody mechanism in BeeGuard is to detect and prevent flooding attacks with scouts by including a immunity property for foreign scouts inside each node.
Whenever a foreign scout appears at a node the immunity against that node is increased. Since the immunity decays at regular intervals normal scouting action won't be affected. But when an unusual high number of incoming scouts is detected, the immunity will rise above a certain level, all further requests will be blocked and not processed anymore.

```
const double IMMUNITY_UPPERBOUNDARY      =1.0f;
const double IMMUNITY_INCREASE           =0.1f;
const double IMMUNITY_DECAY              =1.0f;
const double IMMUNITY_DECAY_INTERVALL    =1.0f;

struct antiBody{
 double immunity;
 int antigen;
 float lastUpdatedAt;
}
```

Listing 11.1: AntiBody Constants and Datatypes

The structure antiBody contains three variables that store the information generated for each incoming scout. The immunity represents the counter to track the scout traffic, the antigen represents the foreign node for which the antibody has been created and lastUpdatedAt helps to improve the update cost as described later and stores the times tamp of the last update of the antibody in seconds of the global simulation time.
There are several constants defined for the AntiBody mechanism that can help to adjust the behavior of it.
The IMMUNITY_UPPERBOUNDARY sets the limit until the immunity of an antibody can increase before incoming scouts of that node are blocked.
IMMUNITY_INCREASE is the amount the immunity value in the antiBody structure is increased each time a foreign scout enters the node.
The last two constants provide the amount and the interval of the regular immunity decrease.

```
class BeeGuard {
public:
    bool increaseImmunity(int node);
    ...
private:
    void immunityDecay();
    ...
    list<antiBody*> antiBodyList;
    ...
};
```

Listing 11.2: AntiBody Methods

There are two functions and one data structure needed to realize the AntiBody system. The list contains all antibodies created at a local node for incoming foreign scouts. Each time a scout enters the node the *increaseImmunity(int node)* is called from the entrance class. If the IMMUNITY_UPPERBOUNDARY is crossed the scout is dropped. Otherwise the immunity counter is increased. In case that there is no entry in the AntiBodyList for the incoming scout's node it is created. At the beginning of the increase immunity function *immunityDecay()* is called to perform an update of the regular immunity decay. The decay is the only way to decrease the immunity and should happen in such an interval so that normal traffic isn't affected but unusual high scouting requests are blocked early enough.
Instead of doing a global decrease each time, those values are just updated each time the antibody list is accessed by the following calculation:

```
(*i)->immunity-= IMMUNITY_DECAY*(currentTime-(*i)->lastUpdatedAt)/
                 IMMUNITY_DECAY_INTERVALL;
```

Listing 11.3: Immunity Decay Function

This way no extra overhead is needed to maintain those lists.
Whenever the immunity of a node drops below zero that entry hasn't been updated for a longer time and is deleted.

11.2.2. BeeGuard::HoneyPot

The honey pot is a heuristic to evaluate the difference of mutual traffic between two nodes in order to detect selfish nodes that want another node to forward their traffic but decline to contribute own resources to the network.

Another effect of honey pot is that massive flooding with foragers causes a node also to shut down the connection.

```
const double HONEYPOT_INITAL_LEVEL        =1.0f;
const double HONEYPOT_CONSUMPTION         =0.01f;
const double HONEYPOT_REFILL_QUANTITY     =0.001f;
const double HONEYPOT_REFILL_INTERVALL    =0.01f;

struct honeyPot{
 double honeyLevel;
 int node;
 float lastUpdatedAt;
};
```

Listing 11.4: HoneyPot Constants and Datatypes

The honeyPot structure contains three elements: the honeyLevel, that is an indicator of the quality of the relationship between the current node and another node stored in the second variable. The third element is a counter similar to the one used in the antiBody structure and it's task is to store the time stamp of the last update of a honey pot since each pot content is increased at a regular interval.
That interval is determined by the constants HONEYPOT_REFILL_QUANTITY and HONEYPOT_REFILL_INTERVALL that describes the gap and the quantity of a regular refill. Those regular refills ensure that nodes with a bad rating shouldn't be blocked completely. This is not necessary that a node is selfish if it does not contribute contribute any traffic. Nevertheless, these nodes do receive a smaller priority.
The HONEYPOT_INITAL_LEVEL represents the initial volume of honey in a honey pot. Consequently, each relationship has a certain starting tolerance in this mechanism.
Finally, the HONEYPOT_CONSUMPTION describes the amount of honey that is added or removed to a pot when foragers are forwarded.

```
class BeeGuard {
public:
    bool consumeHoney(int node);
    void storeHoney(int node);
    ...
private:
    void increaseHoneyStock();
    void createHoneyPot(int node);
    ...
    list<honeyPot*> honeyPotList;
};
```

Listing 11.5: HoneyPot Methods

The honey pot is implemented with the help of four functions and one data structure. In the honey pot list all honey pots for the neighbors of a certain node are stored. The *consumeHoney(int node)* function is called from the Entrance class whenever a forager has to be forwarded by a node that hasn't created it.
storeHoney(int node) is called whenever a forager enters the destination node.
The *increaseHoneyStock()* function periodically increases the amount of honey in each pot. Whenever there is an attempt to increase or decrease the honey volume in a node and the corresponding entry is not available in the honey pot list, the *createHoneyPot(int node)* function is used to create a list entry for the specified node.

11.2.3. BeeGuard::ScoutFreeze

Scout freezing is used to the requests of faked scouts blocking the original ones. This is done by freezing scouts which would have been delete in original BeeAdHoc. Their processing is delayed until the necessary route information for the frozen scout is available at the current node.

```
class BeeGuard {
public:
    void freezeScout(Scout* bee);
    Scout* checkFreezeList(int node);
    ...
private:
    ...
    list<Scout*> freezeList;
};
```

Listing 11.6: ScoutFreeze Methods

The implementation of the scout freeze mechanism had been done in two functions.

No special data types are needed and just a list of the BeeAdHoc scout type is created to store all frozen scouts.
In the original BeeAdHoc an incoming scout is checked if already another replica of it visited the node. When such an entry in the seenScout list is found, the scout is simply dropped.
For BeeGuard that part in the Entrance class has been modified in such a fashion that instead of dropping these scouts they are added to the frozen list.
Whenever a scout returns to a node this frozen list is checked and if it contains any scout that is looking for the same destination as that incoming scout, the additional route information is attached to the frozen scout and it's sent back towards it's source. If there are additional copies of that scout they are simply deleted.

11.2.4. BeeGuard::TrafficPattern

This mechanism is used to create patterns that represent the traffic at each pair of nodes that are communicating with each one another. Each node has a pattern that logs the outgoing traffic and a mirror pattern which logs incoming traffic. And of course the same pair of incomming/outgoing pattern exists in both nodes. If there exist differences between a pair of pattern, a route is blocked for further traffic. This last security mechanism of BeeGuard is the most powerful and also the most complex one.

```
const int MAX_NODES_IN_NETWORK   =15;
const int MAX_PATTERN_TOLERANCE =5;

struct trafficPattern{
 long int routeID;
 int destinationNode;
 long int currentChecksum;
 deque<int> storedChecksums;
};
```

Listing 11.7: TrafficPattern Constants and Datatypes

There are two constants used to define the behavior of this mechanism. The first, named MAX_NODES_IN_NETWORK, is used to ease and to speed up the handling of routes and it will be described later.
MAX_PATTERN_TOLERANCE represents the allowed aberration that is tolerated in the used pattern.

The structure trafficPattern contains all variables that could describe a traffic pattern. The routeID is a compact description of all hops of a whole route. The destinationNode is the target node for the routeID. The last two elements describe the pattern that is used.
At the moment, we use a quite simple pattern that is a simple counter that is increased for each outgoing or incoming packet by a certain number. In the list named storedChecksums a history of pattern is stored and which helps to validate a route.

```
class BeeGuard {
public:
    void updateTrafficpattern(Forager* bee, bool invertedRoute);
    void createTrafficpattern(Scout* bee, bool invertedRoute);
    void insertChecksum(Forager* bee);
    void validateChecksum(Forager* bee);
    bool isRouteOnBlackList(Forager *bee);
    bool isBlackListEmpty();
    ...
private:
    ...
    void addToBlackList(long int routeID);
    long int invertRouteID(long int routeID);
    void createPatternEntry(Forager* bee, long int routeID);
    long int calculateRouteID(deque<int> route);
    long int calculateInvertedRouteID(deque<int> route);
    ...
    list<trafficPattern*> trafficPatternList;
    list<long int> blackList;
};
```

Listing 11.8: TrafficPattern Methods

As mentioned in the last paragraph, complete routes are stored in the TrafficPattern mechanism to improve the speed of the single functions. Instead of storing and comparing routes consisting of single hops an ID is computed with the following function:

```
long int BeeGuard::calculateRouteID(deque<int> route){
    int factor      =1;
    long int routeID=0;
    for(int s1=0;s1<route.size();s1++){
        routeID+=route[s1]*factor;
        factor*=MAX_NODES_IN_NETWORK;
    }
    return routeID;
}
```

Listing 11.9: Calculating Route IDs

Since the support of large scale networks wasn't a design challenge for BeeGuard this method can be used in small networks without producing an overflow. Theses techniques have to be modified to take care of large networks.
The *calculateInvertedRouteID(deque<int> route)* function works the same way but calculates an inverted route ID for the same route, to create a mirror pattern.
To save an additional element in the trafficPattern structure, *invertRouteID(long int routeID)* method is used to calculate an inverted route for a given route ID. This is used when a corrupt pattern is detected and not only the route ID available but also the inverted route has to be blocked.
The *updateTrafficpattern(Forager* bee, bool invertedRoute)* is called each time a forager enters or leaves a node and the relating pattern has to be modified.
The *createTrafficpattern(Scout* bee, bool invertedRoute)* instead is used when a scout reaches it's destination or returns home to introduce the new route. Due to the FrozenScout mechanism, it is possible that a scout retrieves a route without being at the destination itself. To cater for this case, new patterns are created during an update. In all of these cases *createPatternEntry(Forager* bee, long int routeID)* is used to add the new entry.
Whenever a forager is created at a node the pattern list is checked for a valid checksum entry that can be appended to the scout. Valid entry means that the pattern entry cannot describe the same route that forager will follow. Checksums generated for a route are always validated over other routes.
Finally, the validation takes place in the *validateChecksum(Forager* bee)* routine when a forager with a legal checksum enters it's destination node. If no checksum had been found that could have been attached to the forager it carries a **-1** instead.
During a validateChecksum call three alternatives are checked:

- If the checksum to be validated is equal to the current checksum of that pattern, then validation is successful and the history of that pattern is deleted.

- The incoming checksum is higher than the current checksum. The difference of both is checked and if it is below the tolerance range defined by MAX_PATTERN_TOLERANCE then the situation is not alarming.
- The checksum is smaller than the current checksum. In this case the checksum history of that pattern is examined. If the desired entry is found all entries older than the given one are deleted.

If an incoming checksum can't be validated the route is added onto the black list and the checksums for that route and the related inverted route are set to **-1**. This is done to inform the node at the other end of the route that it is corrupted.
The *isBlackListEmpty()* and *isRouteOnBlackList(Forager *bee)* functions are finally used by the Dancefloor class to delete foragers of infected routes.

11.2.5. Embedding BeeGuard in BeeAdhoc

All BeeGuard methods are combined in one class called BeeGuard. The main interface between this class and BeeAdHoc is of course the BeeAdHoc class Entrance. In that class the complete handling of incoming and outgoing network traffic of a certain node is handled. All extern calls of BeeGuard functions take place in the Entrance class, except the black lists that store forbidden routes detected by the traffic pattern mechanism. It is called from the dance floor where the foragers are kept and thus prevents sending forager using banned routes.
Other classes have been also modified due to extension or hierarchy reasons.
Additional forager properties are added to struct forager in b_beedefinitions.h.
So for the implementation the include file b_beedefinitions.h had to be extend for three additional forager properties:

```
struct Forager {
    ...
    long int checkSum;
    long int routeID;
    int contentChecksum;
};
```

Listing 11.10: Additional Beedefinitons

The variable checkSum allows a forager to carry a checksum of a pattern that has to be validated by the destination node. The checksum has to be transmitted in addition to the routeID.

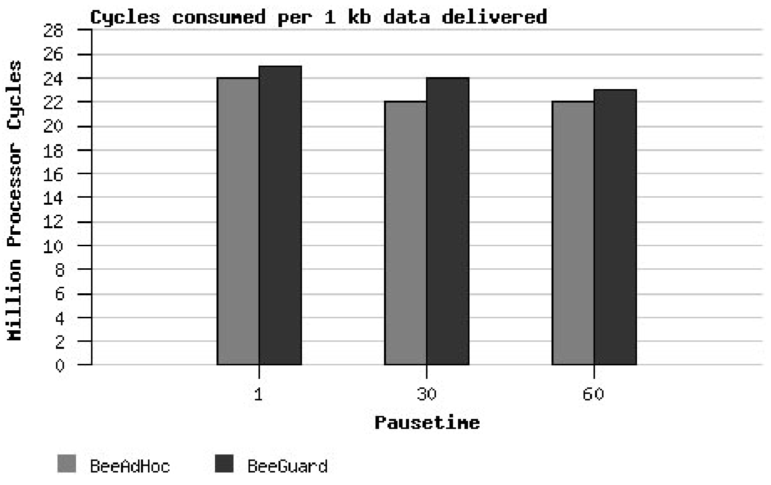


Figure 11.2.: Cyclecounts BeeAdHoc vs BeeGuard

The third new element simulates the checking of a forager's content. In a real application, this value is calculated after analyzing a whole packet including data and routing information.
Since in the simulation environment no real data part is attached to a packet and for simplified testing possibilities this variable describes a packet's content.
The copy forager function in b_dancefloor.h was extended and the hdr_beehive.h was modified.

11.3. Evaluation of the Energy Performance of BeeGuard

To examine the performance overhead of BeeGuard we conducted several test runs for both versions. The environment settings are equal to the ones we used for the profiling tests in the last chapter.
Again we used pause times of 1,30 and 60 seconds, but kept a fixed number of 50 mobile nodes since from our experience appears to be a challenging network.
This time we limited the repetition to calculate an average cycle consumption to two runs since we learned that those cycle result doesn't vary that much.
During all tests we turned off the cache ability of BeeAdHoc.

The results from the cycle measurement represent the amount of cycles needed to deliver one kilobyte of data in the network. As shown in this measurement, BeeGuard needs in all runs less than 7 % of additional processor cycles then BeeAdHoc to maintain a network, as shown in figure 11.2

11.4. Testing the Functionality of BeeGuard

We validated the functionality of BeeGuard by designing different attack scenarios and then evaluated the results.
For each attack scenario we used a modified version of NS-2. Two version includes the standard BeeAdHoc and the other ones contains the BeeGuard extension.
In all versions the same failure behavior for a particular scenario has been implemented and we use one version that logs information about the current attack and the other one for measuring the energy consumption.

Remark

We use some constants for most of our security mechanisms to fine tune their behavior. During setting up test scenarios, we noticed that there is no combination that works for all simulation inputs.
The immunity boundary for example, that defines the limit of allowed scout request in a certain time frame before the node begins blocking them, depends on the packet send rate in the network.
We used values for these constants that fit for our simulation settings, but for a general use of these mechanisms it would be necessary to develop a dynamic approach that sets the behavior in dependency of the network environments.

11.4.1. BeeGuard::AntiBody

Failure Injection

The antibody mechanism should detect high scouting activity from attacking nodes that try to flood the network with these requests. For simulating this behavior we implemented the attackers behavior into two NS-2 versions, one containing BeeAdHoc and one including BeeGuard. Both versions have been modified the same way:

```
int PackingFloor::send(Packeer* packer){
...
if(localaddress==2){
    for(int s1=0;s1<10;s1++)
```

```
        sendScout(rand()\%65000, beehive->INITIAL_TTL, (HDR_IP(packer->
            ns2_p))->flowid());
    }
    ...
    }
```

Listing 11.11: Antibody Failureinjection

This modification causes node 2 to generate 10 additional scouting requests for each real scout demand. The target node is chosen at random each time and the attacker doesn't care about the amount of real nodes in the network.
As test scenario we used an area with a size of 800 x 50 meters containing 10 nodes and a pause time set to 1 second.
In this attack scenario we limited the simulation time to 80 seconds, because with higher durations BeeAdHoc got serious problems.
The malicious node is represented by node 2 as already shown in the code extracts above.
For both protocols we run the scenario three times and so in total we did 6 test runs for this attack scenario.

Results

The results from this experiment are shown in figure 11.3 and 11.4.

- Cycle Consumption
  Comparing the cycle consumption shows the benefit of our security extension: The BeeGuard version needs about 80 % less cycles to deliver 1 kilobyte of data. The reason for this difference is that antibody mechanism decreases the amount of send control packets by about 60.8 % and the number of received packets to 23.14 %. The explanation for this behavior is, that the antibodies prevent the broadcasting of abnormal high numbers of scout request, but cannot prevent the malicious node from creating them. So the number of received control packets is higher than the amount of sent packets.
- Throughput
  BeeAdHoc delivered 2221 kilobyte of data in this attack scenario. The BeeGuard version outnumbered this result by successfully delivering 2780 kilobyte of data through the network. Which is approximately 20 %.

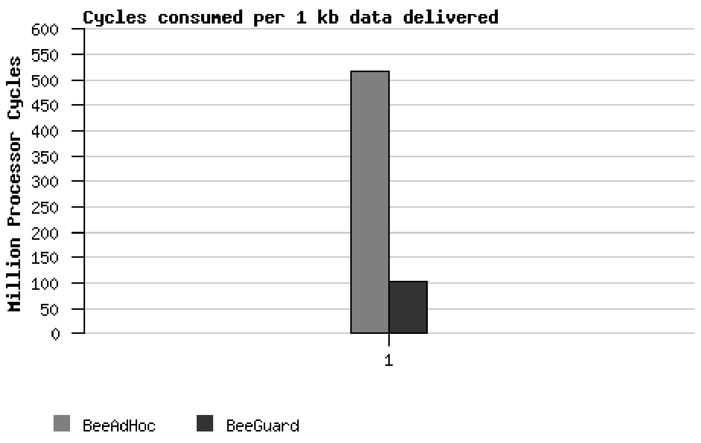


Figure 11.3.: Cyclecounts for Antibodies

Resuming the outcomes of the test results for this function, it reveals that the antibody mechanism improves the security of networks routed with BeeAdHoc. Although this mechanism can't prevent an attacker from sending massive scout requests but is has significantly reduced its impact.

11.4.2. BeeGuard::HoneyPot

Failure Injection

HoneyPots are a rating mechanism for the relationship between two nodes. Its main task is to detect and block selfish nodes, that use the network without contributing of their own resources for maintaining it.
For setting up a specific attack scenario for this mechanism NS-2 has been modified in such a manner that a certain node drops all foreign scouts requests to reduce the number of routes including this node. In this scenario we are interested in the energy consumption of the different nodes in the network as a function of their battery status, especially for the malicious node and also the number of data packets forwarded by each node and the number of packets forwarded in the network that are either created at the bad node or are sent towards it. To receive this information, NS-2 has been modified such that the number of packets forwarded at each node, packets that are sent from or to the attacking node and the current battery state are logged during executing of a simulation.

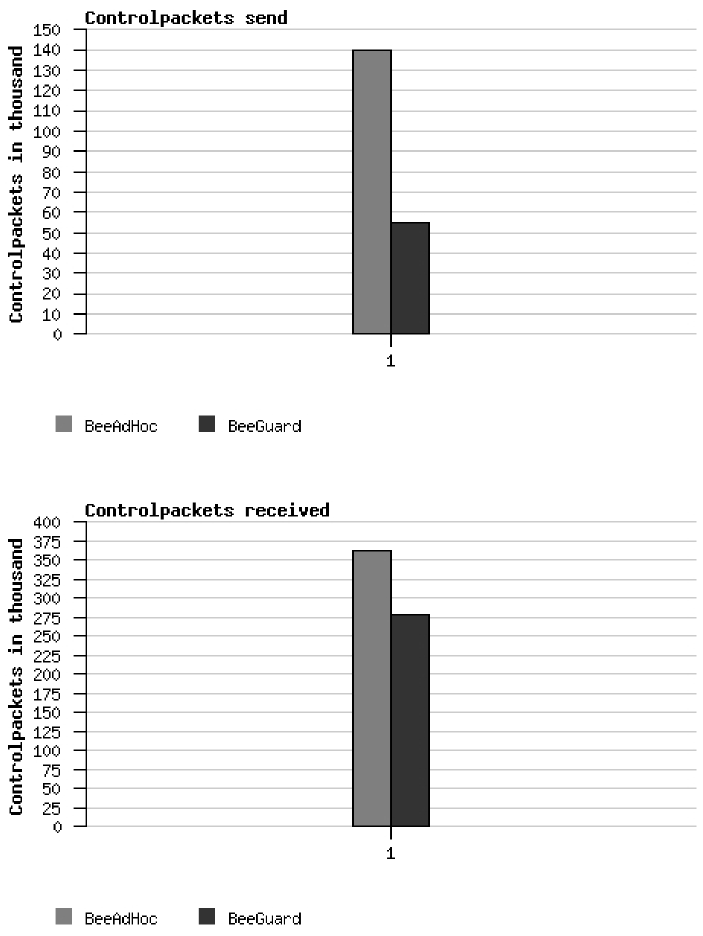


Figure 11.4.: Testing Antibodies: Controlpackets

To simulate the attacking behavior we used the following lines of code:

```
...
    if(localaddress==2&&Scheduler::instance.clock()>0.1){
        beehive->cleanUpScout(bee);
        return;
    }
...
```

Listing 11.12: HoneyPot Failureinjection

Results

In the insecure BeeAdHoc version 0.84 % more energy is consumed during the attack simulation compared with the normal behavior. Due to the BeeGuard security mechanism 18.6 % more energy is left after running the attack scenario, as shown in figure 11.5. The reason for this development is that in BeeAdHoc less packets are forwarded since the attacking node doesn't support the network anymore due to blocked routes. But on the other hand the attacking node isn't busy with forwarding foreign data packets and so able to increase it's own output by 30.31 % data packets, as shown in figure 11.6.
In contrast to BeeAdHoc the BeeGuard version only suffers a lowering of 22.01 % packets that are forwarded in the network and forces the attacking node to decrease it's send rate by 10.98 %.
The cycle count, as shown in figure 11.7 provides the worst result for all of our security mechanisms. BeeGuard uses 37.78 % more cycles in an attack scenario while dealing with a selfish node. There are two reasons for this high number: One the one hand the honey pot rating function has some difficulties in cooperating with the traffic pattern as explained in detail in the final summary at the end of this chapter and also packets that are blocked because of insufficient honey rating consume less cycles while being processed but don't appear in this chart since it displays the ratio of consumed cycles according to successfully delivered data packets. Including this numbers would improve the relationship, for sure.

11.4.3. BeeGuard::ScoutFreeze

Setting up an attack scenario for a rush attack in NS-2 is quite difficult since it requires a very good timing and a detailed setting for the nodes. Because the

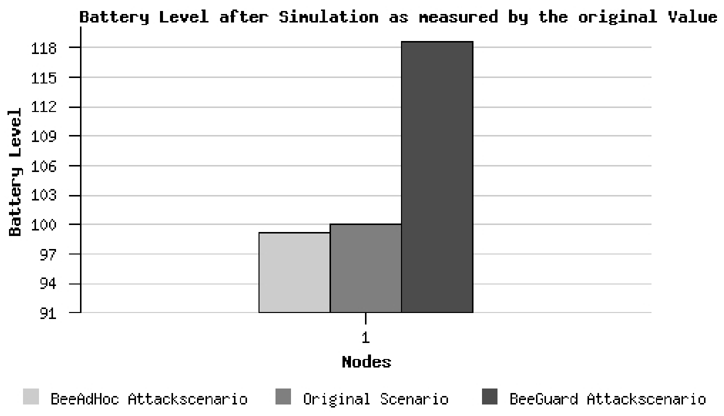


Figure 11.5.: Batterystats for Honeypot Tests

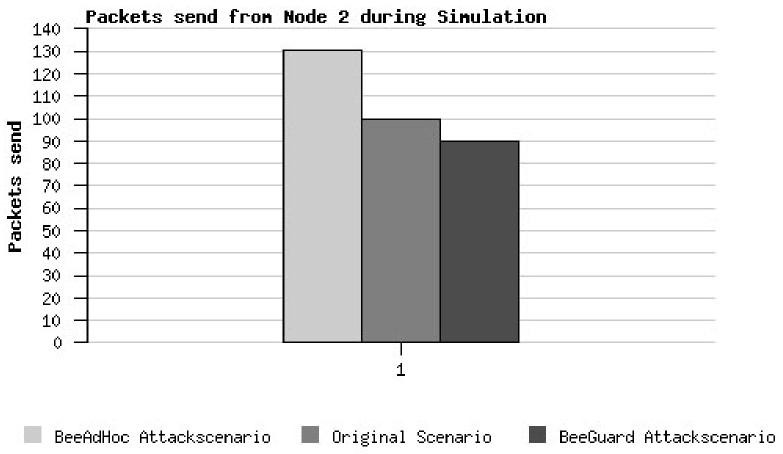


Figure 11.6.: Packets send by attacking Node

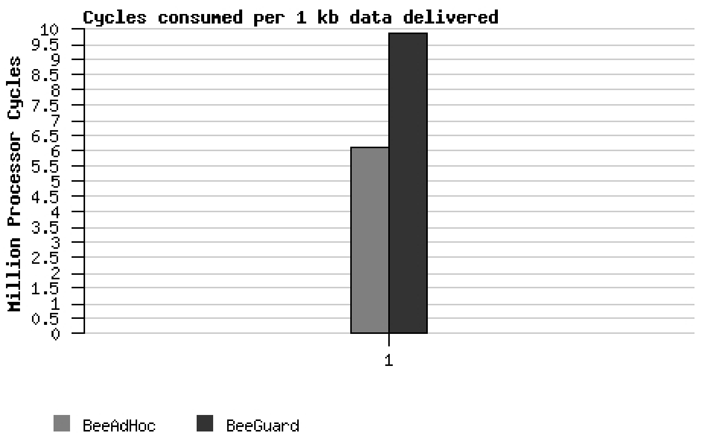


Figure 11.7.: Cyclecount for Honeypot

functionality of this mechanism has been tested in low level tests during the implementation we skipped this test. This mechanism also creates no extra energy consumption in a real attack scenario as compared to normal behavior, since the mechanism simply handles duplicate scout request in a different way. This different scout handling also takes place during normal network behavior and so there is no need to simulate an attack to test it like the other mechanisms.

11.4.4. BeeGuard::TrafficPattern

Failure Injection

The objective of the traffic pattern is to detect changes of a packet's content and route information of forager when they travel in the network.
The focus of this mechanism is guaranteeing the integrity of a packet's content, since most attempts to attack the routing information led to shutdown of that route because of the forager principles. Nevertheless, attacks on route data that aren't prevented by the forager technique are also proceeded by the traffic pattern.

We modified two NS-2 version in the following way:

- One NS-2 version with standard BeeAdHoc has been adjusted, such that in the letIn(Forager* bee) function all packets forwarded by a certain node are considered as corrupted.
  In a real environment, such an attack wouldn't be detected by BeeAdHoc, so it is

not necessary to simulate the insertion of real failures into real data. Just counting the times a corruption would take place is sufficient.

```
void Entrance::letIn(Forager* bee){
...
paketForwarded++;
  if(localaddress==2){
       packetsCorrupted++;
  }

letOut(bee);
...
}
```

Listing 11.13: Trafficpattern Failureinjection 1

Also the packets that are forwarded by a node are counted and logged to allow a better comparison.

- One NS-2 version for BeeAdHoc which is extended by BeeGuard where the *letIn(Forager* bee)* look this way:

```
void Entrance::letIn(Forager* bee){
...
  if(localaddress==2){
       bee->contentChecksum=3;
  }

letOut(bee);
...
}
```

Listing 11.14: Trafficpattern Failureinjection 2

In this version every time node 6 forwards a packet the contentChecksum is set to 3. This variable would contain a checksum generated for the whole packet in a real environment. For testing purposes we set this value to 1 for each packet and every other value is considered as an illegal modification that should be detected by BeeGuard.

As test scenario we use an area with a size of 800 x 50 meters to create a simulation situation where a lot of traffic has to be forwarded over the attacking node.
The environment contains 10 mobile nodes, a pause time is set to 1 second and 400.0 seconds of simulation time.

For both protocols we run the scenario three times to calculate the average cycle consumption. So in total we did 6 test runs for this attack scenario.

Results

Examining the test result shown in figure **??** and 11.9, we concluded the following information:

- Cycle Consumption
  BeeGuard consumes 19.47 % more cycles per one kilobyte of delivered data. This is no surprise since creating and maintaining the blacklist requires additional calculations.
- Corrupted Packets
  In BeeAdHoc the attacking node(2) managed to corrupt 13.472 out of 60,364 packets which is a total of 22,31 %. The last packet has been corrupted at time t=399.131 out of 400.0 seconds of simulation time.
  Of course we chose a node as an attacker that occupied an important position in the network and forwarded a large number of packets to test BeeGuard under massive load.
  In BeeGuard, all routes that include the malicious node had been shut down after 75.0 seconds and the last packet has been corrupted at t=75.2915 out of 400.0 seconds simulation time.
  Until this point in time the attacker only managed to manipulate 51 data packets out of a total of 33093 packets delivered by BeeGuard, which equals a share of 0.15 % data packets.
- Throughput
  BeeAdHoc delivered about 45.7 % more data packets than BeeGuard. One reason for this result is of course that BeeGuard blocks the malicious node and since it occupies a very important position in the network other routes have to be used that are not of such a good quality as the blocked one.
  Taking into account the number of corrupted packets delivered by BeeAdHoc, the additional throughput of BeeAdHoc decreased down to 29.43 % in comparison with BeeGuard.
- Failure Detection
  BeeGuard closed all routes over the malicious node after 51 data packets had been corrupted. There have been four different routes that had to be closed, which makes an average of about 13 corrupted packets per bad route that has to be detected.

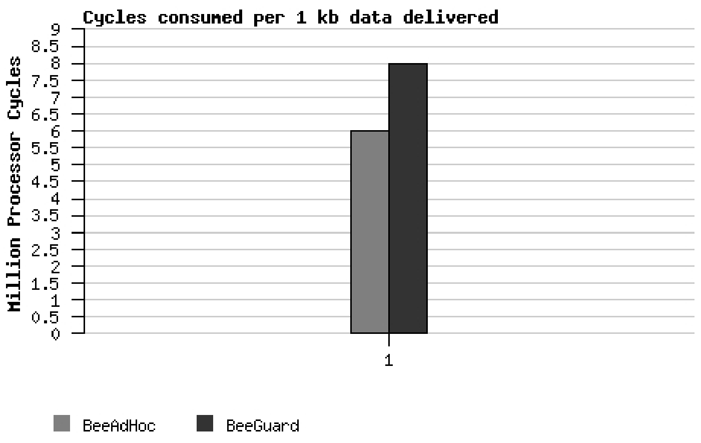


Figure 11.8.: Cyclecount for Trafficpattern

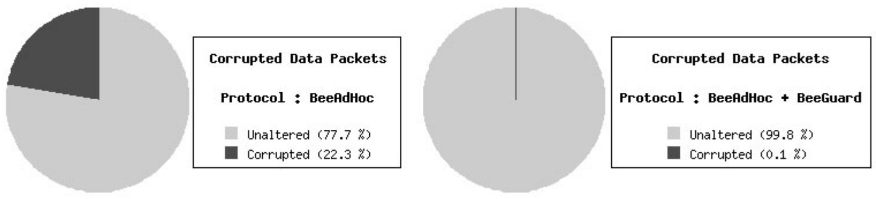


Figure 11.9.: Share of Corrupted Datapackets

The last packet had been corrupted at t=75.2915 for out of 400.0 seconds simulation time and the first corrupted route had been set onto a black list at t=19.144 seconds.

Summarized, BeeGuard shows a good performance. The additional cycle consumption needed for the security mechanisms in this attack situation is in an acceptable cost. The traffic pattern isn't optimized yet, so this performance can be enhanced in future work.

11.5. Conclusion

The results obtained from extensive tests are quite encouraging. The improvements against the different attack scenarios worked the way we planed them and also showed an acceptable cost of an energy consumption. Of course the performance can be better,

but we just delivered the first basic concepts for the security aspect of BeeAdHoc that hasn't been processed so far. We also analyzed the vulnerability of normal BeeAdHoc which showed up the serious security shortcomings of that protocol.
The HoneyPot rating needs to be specially examined since the function works, but it can be counterproductive in combination with the traffic pattern. This happens when a node is in a worse position for a longer time and gets a bad rating this way. Every time a node gets blocked this way a forager is also blocked and the difference between the pattern at the source and the destination node is increased. A few times this can happen without any effect due to the tolerance value we included into the pattern function.
But when it gets blocked for to many times this way, this route will be added into the black list and no further traffic between those two nodes will pass this route. Unfortunately we didn't get aware of this problem during the design and implementation phase and realized it when we launched our test runs. So it was to late to fix until the completion of this report and so this is an important task for the future.
For further developments and improvements, the mechanisms we developed need a more precise adjustment of their parameters. An efficient implementation is an important aspect that we didn't complete so far as well.

12. Routing Framework in Linux

12.1. Overview

After investigating several methods of implementing a new routing algorithm for the Linux kernel we decided to use source routing in combination with the existing Linux Netfilter Architecture [wir05]. The bee type is encoded into the TOS field of an IP

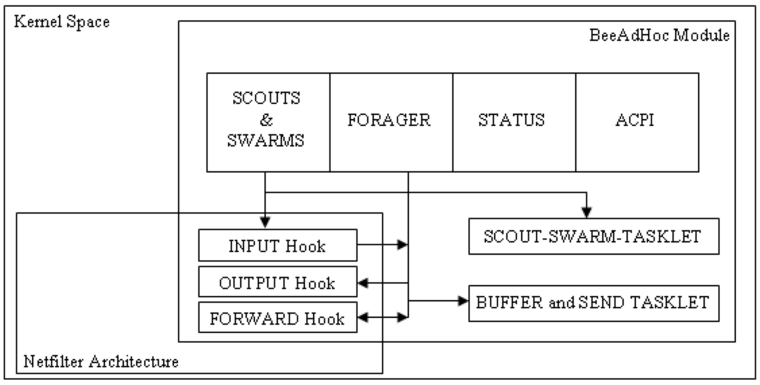


Figure 12.1.: Implementation overview

header . For collecting and transporting bee agents we introduced an RFC-compliant [rfc81] new IP-option.

12.1.1. The Linux Netfilter Architecture

Netfilter ([net]) is the firewalling subsystem of the Linux 2.4/2.6 kernels. Although it's main purpose is to filter packets, but it can also do NAT (network address translation) and packet mangling.

No function present in the netfilter code helped us directly with altering the routing table or decisions of the kernel. However, we utilized five hooks that are already present in the kernel and are called at different locations in the network stack. Kernel modules may register different functions that get called at each of these hooks. Figure 12.2 shows the path of network packets inside the kernel.

```
--->PRE------>[ROUTE]--->FWD---------->POST------>
    Mangle       |       Mangle    ^    Mangle
    NAT (Dst)    |       Filter    |    NAT (Src)
                 |                 |
                 |              [ROUTE]
                 v                 |
                 IN Mangle        OUT Mangle
                 |  Filter         ^  NAT (Dst)
                 |                 |  Filter
                 v                 |
```

Figure 12.2.: Netfilter hooks

The packets in transit enter the kernel on the left in the PREROUTING-hook. After this the kernel makes the routing decision. If the packet is to be delivered locally the INPUT-hook is executed and the packet is delivered to the higher layers of the TCP/IP-Stack. Otherwise the FORWARD-hook is called. If the packet was generated locally it first passes the OUTPUT-hook and is then routed by the kernel. Now its path merges with forwarded packets. Before leaving the system both types of packets are finally processed handed to the POSTROUTING-hook.

12.1.2. Using the IP header for transporting bee data

All additional BeeAdhoc specific data is encapsulated inside the standard IP header of every data packet to achieve maximum compatibility with the existing networks. A standard conform IPv4 header consists of at least 20 bytes, its format is shown in Figure 12.3.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12.3.: Standard conform IP header

IP options

The BeeAdhoc-Routing-Algorithm is based on source routing which means that a packet has all the information required for routing. In addition to this we have to collect routing information from every intermediate node to evaluate the quality of a route according to our requirements.

To realize this with IPv4 we use IP options which are defined in RFC 791 [rfc81]. We utilise the options field which may have a maximum length of 40 bytes and has to be a multiple of 4 bytes. There are two kinds of options:

- Type 1 consists of just one option-type octet: the NOOP (no operation) option and the EOL (end of option list) option.
- Type 2 consists of an option-type octet, a length octet and a variable count of data octets.

The option type octet has three fields: If the copy bit is set, this option has to be

```
Bit   7      6       5      4      3      2      1      0
    +------+------+------+------+------+------+------+------+
    | copy |option class |          option number           |
    +------+------+------+------+------+------+------+------+
```

Figure 12.4.: IP options

copied into each fragment while fragmentation. The classes are 0 (control), 1 (reserved), 2 (debugging and measurement) and 3 (reserved).

Source routing

As the Linux Netfilter Architecture is not intended for routing algorithms, we use source routing to route the packets along their way. The source routes are inserted into (nearly) every packet by the Netfilter BeeAdhoc module (exceptions are broadcast packets). This way the routing table of the kernel consists of just one dummy entry. So all (non-local) packets are sent out through the standard WLAN network interface by the kernel without a need to apply a patch. This can work because we put all computers of the ad-hoc network in the same subnet 10.0.0.0/255.255.255.0. So the routing table of the kernel is shown in Figure 12.5:

As can be seen from this example, it is still possible for a computer to be in several subnets. It does not have to use the BeeAdhoc algorithm for every subnet. If it still has a wired network interface (like in this example) it can use the conventional routing

```
$ route
Kernel IP routing table
Destination  Gateway      Genmask        Flags Metric Ref Use Ifac
10.0.0.0     *            255.0.0.0      U     0      0     0 eth0
192.168.1.0  *            255.255.255.0  U     0      0     0 eth1
default      192.168.1.1  0.0.0.0        UG    0      0     0 eth1
```

Figure 12.5.: routing table

side by side with the BeeAdhoc algorithm. This enables an easy employment of the new algorithm.

There are two types of Source Routing options, loose source routing (type 131) and strict source routing (type 137). We are using the second one because the complete route is calculated at the source host. The first data byte is a pointer which points to the first byte of the next hop address. Route data is composed of a series of IP addresses (four bytes per hop).

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//-+-+-+-+-+
|1 0 0 0 1 0 0 1|   length      |    pointer     | route data     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//-+-+-+-+-+
```

Figure 12.6.: Source route option

To send a source routed packet it has to be created in the following way: The source address has to contain the IP address of the sender, the destination address must not contain the address of the final recipient but of the next hop. The pointer has to point to byte four, the beginning of the route, which is the second hop on the way through the network.

As options data is limited to 40 bytes, we will use three bytes for the source routing option in beginning, 8 x 4 bytes = 32 bytes for the source route, and remaining five bytes for the BeeAdhoc option. That means the longest possible route has a length of 10 hops.

The BeeAdhoc option

For data collection we introduce a new IP option. Our first task was to give our option a name. According to the database mentioned in [Rey02] number 162 (copy, option class 1, option number 2) is free so we were able to (ab)use it for our interests. We decided to give it a length of five bytes because remaining three data bytes are now available for roting information. So we are able to collect three bit of data from every hop, excluding source and destination hop. Our option is structured as follows:

```
byte:  1  |   2  |       3       |      4        |      5        |
   +-+//+-+-+//+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | 162  |   5  |0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|0 0 0|
   +-+//+-+-+//+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     type |length|     |     |     |     |     |     |     |     |
host:               8     7     6     5     4     3     2     1
```

Figure 12.7.: BeeAdhoc option

Host 1 contains the data of its direct predecessor, host 2 the data of its pre-predecessor and so on. To insert it's own data every host has do a three bit left-shift on the 24 bit data field and to insert the value at the very right side (host 1).

12.1.3. Introduction to our implementation

As mentioned earlier we want to use the existing Netfilter Architecture. It offers us the possibility to hook our algorithm into the packet processing chain at different stages: `PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT` and `POSTROUTING` (see fig. 12.2).

Netfilter FORWARD-chain

Because all forwarded packets need to pass the `FORWARD`-chain, we decided to collect our routing relevant data here. Afterwards we return with a `NF_ACCEPT`-code.

Netfilter OUTPUT-chain

The `OUTPUT`-chain is called for locally generated packets. This means we need to enter the source routing information into the header unless we are dealing with a scout packet or a swarm packet. Broadcast packets don't require source routing information in the header, therefore they are not routed anyway. Scouts are using broadcast technique we do not need to process them.

When a data packet is generated, we check the list of stored foragers if a route to the desired destination is already present. If so, we just copy it into the IP header and delete the appropriate forager (unless it is the last one for this destination). The next possibility is that no route information is present. In this case we need to check if we have already sent a scout. If so we need to store the packet for a short time and wait for a scout to return. Then we continue as described in the last paragraph. Otherwise we send out a scout and start a search for aroute to the desired destination.

A TCP connection does not send a lot of packets at once before receiving their acknowledgement from the receiver. This way the number of waiting packets is limited.

But UDP does not wait for acknowledgements before sending the next packet. So packets to different or even the same target might need a significant amount of space.

Netfilter INPUT-chain

The `INPUT`-chain is the last station of a packet before being delivered to higher layers. All packets destined for the local computer pass this hook. This is why we chose to use it to record all important information from the network packets.

Again we ignore broadcast. This way we ignore scouts and swarms as well (see 12.1.3).

Now we are just dealing with regular packets that are delivered through the BeeAdhoc routing algorithm. This means they contain valuable information that needs to be saved. Every packet is regarded as a forager which can find it's way back to the source computer. Of course we don't need to store the whole packet just the bits of information that are important for us. Once extracted we evaluate the efficiency with a rating function and store it in a table hashed with the source address.

13. BeeAdHoc in Linux

13.1. Kernel Diffs

13.1.1. Necessary modifications inside the original Linux kernel code

Our goal was to modify the original kernel code as little as possible but a few modifications were necessary to fully realize our new BeeAdHoc option.

Modifications inside /include/linux/ip.h

Diff file 13.1 shows two passages out of the kernel header file `ip.h`. In the first part we see definitions of different IP header options. We just added the definition of our BeeAdHoc option according to the description in Section 13.2.

The second part is inside the IP header data structure (`struct iphdr`). It contains kind of pointers (unsigned char) to the number of the octet inside the header in which one particular option starts. These pointers can contain values between 21, which points to the first octet inside the options field - see 12.3, and 40, which points to the last octet inside the options field. We just added a pointer to the beginning of the BeeAdHoc option here.

```
--- linux-2.6.7-orig/include/linux/ip.h
+++ linux-2.6.7-beeadhoc/include/linux/ip.h
@@ -60,6 +60,7 @@
 #define IPOPT_SID       (8  |IPOPT_CONTROL|IPOPT_COPY)
 #define IPOPT_SSRR      (9  |IPOPT_CONTROL|IPOPT_COPY)
 #define IPOPT_RA        (20|IPOPT_CONTROL|IPOPT_COPY)
+#define IPOPT_BEEADHOC   (2  |IPOPT_RESERVED1|IPOPT_COPY)

 #define IPVERSION       4
 #define MAXTTL          255
@@ -100,6 +101,7 @@
                ts_needtime:1,              /* Need to record
                   timestamp           */
```

```
                  ts_needaddr:1;               /* Need to record
                     addr of outgoing dev  */
   unsigned char router_alert;
+  unsigned char beeadhoc;
   unsigned char __pad1;
   unsigned char __pad2;
   unsigned char __data[0];
```

Listing 13.1: Changes in ip.h

Modifications inside /net/ipv4/ip_options.c

This patch is inside the function `ip_options_compile(..)`, which is responsible for checking the correctness of IP options in incoming packets, detecting all known IP options and for filling the option pointers (see above) inside `struct iphdr` with correct values. As the beeadhoc option is new we had to integrate finding it into this function as you can see in patch 13.2.

```
--- linux-2.6.7-orig/net/ipv4/ip_options.c
+++ linux-2.6.7-beeadhoc/net/ipv4/ip_options.c
@@ -433,6 +433,14 @@
                          if (optptr[2] == 0 && optptr[3] == 0)
                                  opt->router_alert = optptr - iph;
                          break;
+                       case IPOPT_BEEADHOC:
+                          if (optlen < 3) {
+                                  pp_ptr = optptr + 1;
+                                  goto error;
+                          }
+                          opt->beeadhoc = optptr - iph;
+                          break;
                        case IPOPT_SEC:
                        case IPOPT_SID:
                        default:
```

Listing 13.2: Changes in ipoptions.c

Adding and modifying ipv4options match

We took the ipt_ipv4options match (`ipt\_ipv4options.c`, `ipt\_ipv4options.h`) from the netfilter extensions and added it to the kernel patch to simplfy netfilter rules. We slightly modified it to match against the BeeAdHoc option. Of course the Makefile and Kconfig were adapted to be able to compile the new kernel.

13.1.2. Optional modifications inside the original Linux kernel code

In addition to this we significantly modified the Linux ACPI code to be able to collect energy data for evaluating the quality of a route based on this metric. We had to do it because all battery values have been declared static inside `/drivers/acpi/battery.c` and the only way to retrieve the current battery state was to read it from procfs. We did only changed accessibility of the needed data. For people who do not like to work with modified ACPI code (and especially for UML which does not have real ACPI code) we made this patch optional. A machine without this patch or even without any ACPI support at all will be able to run inside a beeadhoc network without any restrictions. The point is that such a machine will allways claim that its battery is full.

13.2. Kernel Module

13.2.1. The kernel module

For this to work, we need to write the target (in this case `BEEADHOC`) as a kernel module named `ipt_BEEADHOC.c` and a shared library for the `iptables`-frontend.

To become as independent as possible from the actual kernel version we did not work inside the kernel source tree but rather in an extra directory. That gives us the opportunity to work with the latest kernel version without having to modify our code (with exception of the kernel patches).
We divided the kernel module code into several units [wir05]: The module specific code, like initialisation and module unloading as well as the packet handling code can be found in `ipt_BEEADHOC.c`. Data structures and constants are defined in `ipt_BEEADHOC.h`, methods for procfs output `ipt_BEEADHOC_stat.c`. ACPI functionality for reading the battery state lies in `ipt_BEEADHOC_acpi.c`.
Also there is a file for scouts called `ipt_BEEADHOC_scouts.c` and one for swarms called `ipt_BEEADHOC_swarms.c`. The last file is the for forager `ipt_BEEADHOC_struct.c`.
All files are joined together by `#include<>` directives inside `ipt_BEEADHOC.c`.

We start with the description of three functions that implement the core functionality of a netfilter module.

These functions are:

- `static int __init init(void)`
- `static void __exit fini(void)`

-
  ```
  static int ipt_beeadhoc_checkentry(const char *tablename,
  const struct ipt_entry *e,
  void *targinfo,
  unsigned int targinfosize,
  unsigned int hook_mask)
  ```

The first two are the initialisation code which is executed by the kernel on loading or unloading the BeeAdhoc kernel module. The macro `__init` shows the kernel that this function can be unloaded from memory once the module has been loaded correctly. For initialisation a `struct ipt_target` is filled with function pointers and module name and registered with the kernel in `init()`. Inside these functions we call the initialise or rather the tidying up functions of our submodules.

The third function `ipt_beeadhoc_checkentry` is called every time a netfilter rule is inserted with the `BEEADHOC`-target. It can make some sanity checks whether this rule is to be accepted or not. To test this functionality the module verifies that the rule is intended for the `mangle`-table. Otherwise it prints an error message and returns an error code, so the rule is not inserted into the specified netfilter table.

IP datagram manipulation is done inside the `ipt_beeadhoc_target` function which we describe after handling the internal data structures and the API.

13.2.2. The module core: ipt_BEEADHOC.c

As mentioned before all threads run together inside `ipt_BEEADHOC.c`. It encapssulated the module functionality, the packet mangling and some other helping functions.

Concurrency

Ordinary netfilter code does not include concurrency. A packet is put into a chain and leaves it at the end if it is not dropped or rejected. Because we have to queue packets while waiting for a route there is a need for events happening almost parallel to the standard packet handling thread. We used tasklets in our solution. Tasklets are a form of soft IRQ suitable for small tasks. If scheduled by the kernel, their code is executed once, after the hardware interrupt service routines are finished next time.

Our two tasklets serve the following purposes:

- whenever a scout comes in, the `queue_tasklet` runs through the `bufferqueue` and tries to build packets containing the destination address of the incoming scout. If a packet is ready to send out, it is sent direct by this tasklet.
- everytime the module does not have a matching route to the destination, the scout tasklet is scheduled which sends scouts by broadcast technique to the destiantion and stores the data packets into the bufferqueue.

Core initialisation

As in every submodule we have to do some initialisation stuff inside the core. Inside `static int __init beeadhoc_tasklet_init(void)` we initialise the tasklets, a linked list, and create a socket which will be used later for sending scouts and swarms. The linked list is the `bufferqueue`. We use this queue for queueing packets while they are waiting for a scout.

The ipt_beeadhoc_target function

The core functionality is implemented in the `ipt_beeadhoc_target()`-function. This function is called with a pointer to a sk_buff (see 13.1). A sk_buff is a data region and a collection of pointers (see fig. 13.1). Its pointers point to the network headers of different layers in the array. This way the payload can be copied into the region. Network headers of different layers can now be copied in front of (and eventually after) the payload data without moving or copying it.

This sk_buff also contains a pointer to a structure of IP option, but we cannot use this. The struct is filled while the packet is generated by the kernel. Once the packet is created the data section of the sk_buff is already filled with the RFC-compliant IP-header directly followed by the payload data.

Independent of the hook from which the function is called, the first action is to recognise the type of bee it is actually handling from the IP header TOS field.

```
        __u8 tos = iph->tos;
        __u8 bee_type = IPT_BEEADHOC_OPT_ENERGY;
        if (iph->tos == IPTOS_LOWDELAY){
                ownership = IPT_FORAGER_OWN;
                bee_type = 2;
                }
       if (iph->tos == IPTOS_MINCOST){
                ownership = IPT_FORAGER_FOREIGN;
            bee_type = 2;
```

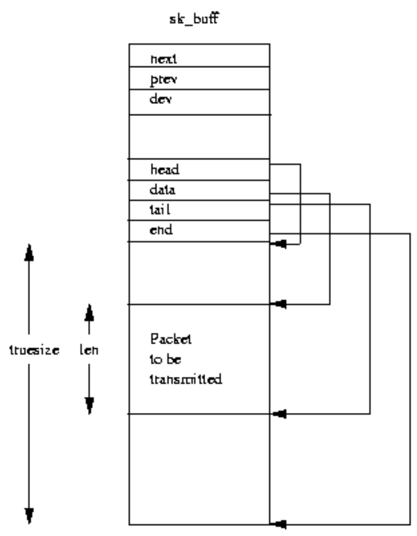


Figure 13.1.: sk_buff

```
          }
```

After getting the bee type we branch depending on the hook. The most interesting case is an outgoing packet which means we are inside the OUTPUT chain.

Netfilter OUTPUT-chain

The main code in the output chain is really compact. It asks the data structure for a forager to the matching destination address with a correct bee type. If it gets such a forager it is able to build the route into the packet by the function `ipt_beeadhoc_build_packet(forager, sk_buff)` and to return NF_ACCEPT. This should be the most usual case, so that we do not produce significant processing overhead in our code (we will show you the overhead of the build function later).

If we do not get back a suitable forager we set `nextScoutDest` to the destination address, schedule the `scout_send_tasklet`, store the corresponding sk_buff in the buffer-queue and report it as `NF_STOLEN`. In this way we direct this packet towards our module for further processing.

```
          struct forager *my_forager ;
          if (beeadhoc_forager_get(iph->daddr, bee_type,
              IPT_BEEADHOC_FORAGER_REMOVE, &my_forager,IPT_FORAGER_OWN)
              == -1) {
                    if (beeadhoc_forager_get(iph->daddr, bee_type,
                        IPT_BEEADHOC_FORAGER_REMOVE, &my_forager,
                        IPT_FORAGER_FOREIGN)==-1){
                    nextScoutDest = iph->daddr;
                            if (speicherbar(iph->daddr)==0){
                                    send_scout(iph->daddr, iph->saddr);
                                    tasklet_schedule (&send_scout_tasklet
                                        );
                            }
                    }
          }
```

Listing 13.3: Output chain code

The `scout_send_tasklet` uses the socket, created in the init phase, to send out a scout packet for the address, contained in `nextScoutDest` to the destination. Once a scout comes back the `queue_tasklet` is scheduled.

Functionality of the `queue_tasklet` is wrapped inside a loop over the bufferqueue. As it knows a scout came in, carrying a route to the address stored in `LastForagerDest`, it takes every packet having that destination out of the bufferqueue and does nearly the same as the OUTPUT-chain code then: It tries to get a forager and puts the route into the packet. If a packet cannot be built because of a missing forager a new scout will be sent out and the loop breaks - no more packets to the particular destination can be sent. The difference to the OUTPUT-chain code is that we are outside the standard thread; every packet has to be sent manually.

```
static void QueueTaskletFunction (unsigned long data)
{
          .
          next_sk_buff = temp_sk_buff->next;
          iph = temp_sk_buff->nh.iph;
          if (iph->daddr==ip) {
                    if (beeadhoc_forager_get(..., IPT_FORAGER_OWN)== -1)
                        {
                            if (beeadhoc_forager_get(...,
                                IPT_FORAGER_FOREIGN)== -1){
                                    nextScoutDest = iph->daddr;
                                    if (speicherbar(iph->daddr)==0){
                                            send_scout(iph->daddr, iph->
                                                saddr);
                                            tasklet_schedule(&
                                                send_scout_tasklet);
                                    }
                            }else{
                                    skb_unlink(temp_sk_buff);
                                    ipt_beeadhoc_build_packet(...,
                                       IPT_FORAGER_FOREIGN);
                                    temp_sk_buff->nfcache |= NFC_ALTERED;
                                    ip_route_me_harder(&temp_sk_buff);
                                    dst_output(temp_sk_buff);
                            }
                    }else{
                            skb_unlink(temp_sk_buff);
                            ipt_beeadhoc_build_packet(...,IPT_FORAGER_OWN
                               );
                            temp_sk_buff->nfcache |= NFC_ALTERED;
                            ip_route_me_harder(&temp_sk_buff);
                            dst_output(temp_sk_buff);
                    }
          }
          temp_sk_buff = next_sk_buff;
          .
}
```

Listing 13.4: QueueTaskletFunction - Read the queue and build packets

It removes the first sk_buff from the sendqueue and instructs the kernel to reroute it. This has to be done because we manipulated the destination address. While rerouting the kernel for example looks up the correct MAC address for the next HOP. Now the packet is sent out.

The central function - invoked by the OUTPUT-chain code as well as by the `queue_tasklet` code is the function `ipt_beeadhoc_build_packet()`. Its job is to take the source route from the given forager and to put it into the IP header of data packet. In addition to this it has to add the BeeAdhoc option exactly there too. The main problem was to figure out the proper way to add the IP options. Unfortunately we cannot be sure that there is enough room free in front of the transport layer header to move it to the front and add our IP options afterwards. So we need to make some room in front of the IP header if necessary. This is done with a call to the function `skb_cow(struct sk_buff *skb, int headroom)`. (The parameter `headroom` is the amount of bytes to reserve in front of the data area. If the headroom is already large enough this call does nothing.) Now we have to `memmove` the IP header to the beginning of the buffer, so that exactly the number of bytes we need between the header and the payload data is freed. After changing some pointers so that the new beginning of the network header is known and after changing the length of the packet and the IP header, we can go on filling the freed space with our own options.

```
opt = &(IPCB(my_skb)->opt); // pointer to options array
if (opt)
        oldoptlen=opt->optlen;
inclen = (route_length>1) ? IPT_BEEADHOC_OPTION_LENGTH
        + route_length*4 - 1 : IPT_BEEADHOC_OPTION_LENGTH ;
fillen = (4 - (inclen + oldoptlen)%4)%4;
inclen+=fillen;
iphlen = iph->ihl + (inclen >> 2);

/* testing, if options length is still OK*/
if (iphlen > 15) {
        printk( "Sorry. Headerlength %i is too big \n", iphlen);
        return -1;
}
if (skb_cow(my_skb, 60 - iphlen)) {
        DEBUGP("skb_cow failed\n");
        return -1;
}
new = skb_push(my_skb, inclen);
if (opt) {
        memmove(new, new + inclen, sizeof(struct iphdr) +
                                        opt->optlen);
} else {
        memmove(new, new + inclen, sizeof(struct iphdr));
}
```

Listing 13.5: creating space for IP options inside an sk_buff

If the length of the route is equal to one, which means we are sending data to our neighbour host, we just have to include the BeeAdHoc option. If it is greater than one we have to include the BeeAdHoc option and source routing information into the header. Of course we have to manipulate the destination address in this case. To achieve options length to be a multiple of four we will add padding in form of NOOPs as well.

```
ipopts[0] = IPOPT_BEEADHOC;
ipopts[1] = IPT_BEEADHOC_OPTION_LENGTH;
ipopts[2] = 0;   // BeeAdhoc data
```

```
ipopts[3] = 0;  // BeeAdhoc data
ipopts[4] = 0;  // BeeAdhoc data

if (route_length>1) {
        ipopts[5] = IPOPT_SSRR;
        ipopts[6] = (route_length*4)-1;
        ipopts[7] = 4; /* pointer */

        /* set destination address to next hop: */
        iph->daddr = route[1];
        opt->faddr = route[1];

        /* insert hops into ipopts */
        for (count=2; count<=route_length; count++) {
                int* test = (int*)&
                        ipopts[IPT_BEEADHOC_OPTION_LENGTH
                        +3+(count-2)*4];
                *test=route[count];
        }
}

/* fillup options with NOOPs to reach  length % 4 = 0 */
for (count=0; count<fillen; count++){
        ipopts[inclen-1-count]=IPOPT_NOOP;
}
```

Listing 13.6: Inserting IP options

Netfilter FORWARD-chain

Inside the FORWARD-chain we put relevant information about the hop into each bee, arriving at the node, dependending on the bee type. In contrast to the simulation we are only able to collect energy data in reality. Energy data, represented by the remaining battery capacity, stored in `ipt_beeadhoc_battery_status`, is put into the IP packet by the `energy_check` function.

For mapping the real data to a three bit value we use the mapping function similar to the one inside simulation. Next step is to read the value of the existing BeeAdHoc option. As we are not able to work on 24 bit values directly (remember: the BeeAdHoc option data field consists of 24 bit, three bit for each host), therefore, we additionally take the length octet and put these four bytes into an __u32, which has to be converted from networking to host byte order. After the length octet is stored in an extra variable, because we have to restore it, we just shift beedata left about three bit and put the actual battery data of our host into the very right three bit. Bee data can now been written into the IP packet, the procedure is finished after restoring the length octet. Handling of other data, like delay or throughput data, works in a similar way.

```
static void energy_check (struct sk_buff *skb)
{
        int batStand = ipt_beeadhoc_battery_status;
        unsigned char batLevel=0;

        if (batStand <= 9) batLevel = 0;
        else if (batStand > 9 && batStand <= 14) batLevel = 1;
        [.....]
        else if (batStand > 65) batLevel = 7;

        struct ip_options *opt;
        opt = &(IPCB(skb)->opt);
        unsigned char* bee_ptr = (unsigned char*) skb->nh.iph +
                        opt->beeadhoc +1;
        __u32* bee_data = (__u32*) bee_ptr;
        unsigned char saveme = *bee_ptr;
        __u32 beedata = ntohl(*bee_data);
        beedata = beedata <<3;
        beedata |= batLevel;
        *bee_data=htonl(beedata);
        *bee_ptr= saveme;
}
```

Listing 13.7: Energy check function

Netfilter INPUT-chain

In `LOCAL_IN` we have to produce a forager from every incoming IP packet. First of all we extract the bee data field from the IP options. As we have to extract a 24 bit data field we have to do some shifting to get the correct value. If the source of the packet is a neighbour, we can create a forager by containing bee data and the source address of the IP packet. Otherwise we have to extract source routing information and add them to the forager. The next step is to add the forager, together with the bee type (see above, TOS field) to the data structure by using `beeadhoc_forager_put(newForager, bee_type, owner)`.

We noticed that most applications answer source routed packets by using the reverse source route on the way back. To prevent this we will have to delete source routes from all incoming packets and fill up the options with NOOP options.

Using ACPI

If ACPI is deactivated `ipt_beeadhoc_battery_status` will allways pretend the remaining battery capacity to be 100%. But if the kernel is patched (see 13.1.2) and the system is capable of using ACPI we are able to read the correct battery state at every moment.

All implementation details about using the ACPI battery state can be found in `ipt_BEEADHOC_acpi.c`. During initialisation the battery handle has to be found. We use `acpi_get_devices` for this task. The first parameter is the internal name for ACPI batteries, the second is the name of a little call back function doing nothing but putting the handle of the first found battery into the battery struct, which is handed over as a third parameter. Now we are able to access the actual battery data. For reading the remaining battery capacity regularly we use a kernel thread. There are two reasons for this: a) we can do the reading independent from all other iptables code and b) we cannot access battery data from the interrupt context (taking a kernel timer would have been our choice then). Starting the kernel thread is the last thing which is done during initialisation of this submodule.

```
static int __init
BeeAdHoc_acpi_init (void)
{
// Battery HID is PNP0C0A
        battery = kmalloc(sizeof(struct acpi_battery), GFP_KERNEL);
        // Look for acpi_battery
        acpi_get_devices("PNP0C0A", battery_probe, battery , NULL);

        init_waitqueue_head(&wq);
        ThreadID=kerneBatteryWatcherFunctionrFunction, NULL,
           CLONE_KERNEL);
        if (ThreadID==0)
                return -EIO;
        return 0;
}
```

Listing 13.8: ACPI initialisation

A kernel thread is handled like an ordinary userspace process (you can even see it inside the process table) but it is able to access every kernel data structure. After it started it is daemonized by putting it into the background. Otherwise the insmod process would be blocked until the thread dies. To be able to kill it - and to unload the module - sending `SIGTERM` has to be allowed. As we do not want to run it all the time and to block everything else, the thread is sleeping for most of its lifetime on a waitqueue. Every five seconds it wakes up, puts the remaining capacity into `ipt_beeadhoc_battery_status` and sleeps again. It does so until it catches a `SIGTERM`.

```
static int BatteryWatcherFunction( void *data )
{
        daemonize("BatteryWatcher");
        allow_signal( SIGTERM );
        while (TRUE) {
                timeout=HZ * 5;
                timeout=wait_event_interruptible_timeout( wq, (
                    timeout==0), timeout );
                acpi_battery_get_status( battery, &bat_stat );
                acpi_battery_get_info( battery, &bat_info );

                int bat_max = (int) bat_info->design_capacity;
                int energy = (int) bat_stat->remaining_capacity;

                if (bat_max == 0){        // no battery -> power from
                    line
                        ipt_beeadhoc_battery_status =
                            IPT_BEEADHOC_BATTERY_FULL;
                } else {
                        ipt_beeadhoc_battery_status =
                            IPT_BEEADHOC_BATTERY_FULL
                                  * energy / bat_max ;
                }
                if (timeout==-ERESTARTSYS ) {
                        ThreadID=0;
                        break;
                }
        }
        complete_and_exit ( &OnExit, 0);

}
```

Listing 13.9: Battery watcher kernel thread

13.3. Datastructs

Structures

All C structs are defined in the header file ipt_BEEADHOC.h.

The most important data structure in the kernel is the hash table.

```
struct list_head *forager_hash;
```

Listing 13.10: Main hash table.

It is an array of size HASHSIZE (currently 256) entries. Every entry is the head of a linked list. In these linked list entries of the form struct daddr_list_entry are stored.

```
struct daddr_list_entry {
        struct list_head list;

        __u32 daddr;
        int count[IPT_BEEADHOC_OPT_MAX];
        int countsum;
        unsigned long timestamp;
        struct forager* forager[IPT_BEEADHOC_ARRAYSIZE *
           IPT_BEEADHOC_OPT_MAX];
};
```

Listing 13.11: daddr_list_entry struct

For every destination IP with stored foragers one of these struct daddr_list_entry exists. They contain the destination IP for quick reference and housekeeping information about the stored foragers. countsum is the total amount of foragers stored of any type whereas count[] is set to the number of foragers of the given type (energy, delay, throughput). The member timestamp helps for the garbage collection when overaged entries are removed. The last member is the actual array containing pointers to the stored struct forager.

```
struct forager {
        struct source_route *route;
        __u32 daddr;
        unsigned long  timestamp;
        __u32 info;
        __u8 nr_dances;
};
```

Listing 13.12: Forager struct

These structs contain all the information we collect from the foragers that "land" on our node: Source route, destination address (or from the point of the landed forager it's source address), timestamp of its arrival, information regarding energy, throughput or delay and the number of dances used to recruit new foragers.

The `struct source_route` is just another linked list used to record all hops of the taken route.

```
struct source_route {
        struct list_head list;

        __u32 hop;
};
```

Listing 13.13: Source route struct

The BeeAdHoc-API

To ease the programming and provide consistent locking an API was implemented. It consists of a few calls to aid in modifying the above structures.

- static int beeadhoc_forager_get(__u32 daddr, __u8 opt_type, __u8 remove, struct forager** forager);
- static int beeadhoc_forager_put(struct forager *forager, __u8 opt_type);
- static void beeadhoc_struct_free_forager(struct forager*);
- static struct forager* beeadhoc_struct_copy_forager(struct forager* forager);
- static void beeadhoc_struct_debugp_forager(struct forager* forager);
- static void beeadhoc_struct_debugp_hash(void);

All these calls are implemented in a single source file (`ipt_BEEADHOC_structs.c`).

On loading the module the function `beeadhoc_struct_init()` is called. It allocates the memory for the hash table and initializes the linked list for every entry.

On unloading the function `beeadhoc_struct_fini` is called. It calls the garbage collection `beeadhoc_struct_gc` with a maximum age of 0 seconds. This way all entries are removed as they are considered out-dated. Afterwards the hash table is freed.

beeadhoc_forager_get

The first essential call. It returns a `struct forager` (passed by reference) for the given `daddr`. If available a forager of type `opt_type` is returned (`IPT_BEEADHOC_OPT_ENERGY`, IPT_BEEADHOC_OPT_DELAY or IPT_BEEADHOC_OPT_THROUGHPUT). The option remove indicates if the forager should be deleted if appropriate (IPT_BEEADHOC_FORAGER_REMOVE or IPT_BEEADHOC_FORAGER_KEEP). Even if IPT_BEEADHOC_FORAGER_REMOVE is passed the forager is not necessarily removed from the kernel. If the route was good and it is still young enough it might dance and in this case a copy is made and returned instead of the original forager.

The return value indicates how many foragers are left for this destination address. If −1 is returned no forager was found and a scout should be sent out.

beeadhoc_forager_put

The second essential call. As the name suggests it inserts a struct forgager into the data structure. The option opt_type indicates the type of the forager to insert (IPT_BEEADHOC_OPT_ENERGY, IPT_BEEADHOC_OPT_DELAY or IPT_BEEADHOC_OPT_THROUGHPUT). For a returned scout this call should be made three times with the three different opt_types. Care should be taken not to make this call with the same struct forager* pointer, but instead with copies of the original forager. beeadhoc_struct_copy_forager can help with this.

Before inserting the forager the garbage collection will eventually be called. The probability of these calls can be influenced by the constant IPT_BEEADHOC_GARBAGE_PROBAB. All destination addresses without traffic in the last IPT_BEEADHOC_MAXAGE seconds are removed and the used memory is freed. There is also a timer running that will free all foragers after IPT_BEEADHOC_MAXAGE seconds.

The return value indicates how many foragers are left for this destination address and opt_type.

beeadhoc_struct_free_forager

As the struct forager contains a linked list, all items need to be freed. To ease this job a simple call to beeadhoc_struct_free_forager is required.

beeadhoc_struct_copy_forager

For the same reason mentioned above copying foragers should be done to this call. It copies all items of the struct source_route-list and returns a pointer to an identical forager.

beeadhoc_struct_debugp_forager

As the name suggests this function is for debugging purposes. It prints a forager with all its information into the kernel log.

beeadhoc_struct_debugp_hash

This is also a debugging function. It iterates over the complete hash table finding all destination addresses and prints information regarding the age of the entries and also about every forager. This call should be used sparingly (preferably inside an `#ifdef`-statement) as it clutters the screen and the kernel log.

Helper functions

The API-calls depend on a number of helper functions. The most important ones are explained below. These are not meant to be called directly. They are just included to help understand the implementation of the API-functions.

beeadhoc_struct_gc

As the name suggests this is the garbage collection function. It loops through the whole hash table and examines every `struct daddr_list_entry`. If the entry is considerd too old (compared to the passed option `maxage`) it and all its foragers are deleted.

```
/*
 * Do Garbage collection.
 * Cycle through hash table and remove all daddr_list entries
 * unused for maxage seconds
 */
static void beeadhoc_struct_gc(unsigned int maxage)
{
        static unsigned int i;
        static struct daddr_list_entry *daddr_list, *n;
        static unsigned age;

        WRITE_LOCK(&beeadhoc_hash_lock);
        for(i = 0; i < IPT_BEEADHOC_HASHSIZE; i++) {
                list_for_each_entry_safe(daddr_list, n,
                                                 &forager_hash[i],
                                                    list) {
                        age = get_seconds() - daddr_list->timestamp;
                        if(age >= maxage) {
                                remove_daddr_entry(daddr_list);
                                list_del((struct list_head*)
                                    daddr_list);
                                kfree(daddr_list);
                        }
                }
        }
        WRITE_UNLOCK(&beeadhoc_hash_lock);
}
```

Listing 13.14: beeadhoc_struct_gc

forager_rate_lifetime

This function determines the value of `nr_dances`. It does not set the value directly but instead returns the correct value.

First of all the length of the source route is calculated. If the destination is a direct neigbour the forager gets a maximum rating as there is no better route.

If this test fails the information gathered by the forager is examined. Every hop stores a 3-bit value in the info field. This is extracted and put into an array. From this array the minimum and the average is calculated. Based on these numbers an appropriate value is returned.

```
/*
 * Look at info and decide how often the forager should dance.
 * Direct neighbours get a maximum rating.
 */
static int forager_rate_lifetime(struct forager* forager)
{
        static unsigned int route_length;
        static __u32 info;
        static unsigned short values[IPT_BEEADHOC_MAXHOPS];
        static unsigned short i;

        route_length = get_route_length(forager);

        if(route_length < 2)     /* Direct neighbour */
                return IPT_BEEADHOC_MAX_DANCES;

        info = forager->info;
        for(i = 0; i < route_length; i++) {
                values[i] = info & 7;
                info = info >> 3;
        }

        unsigned short min = 7;
        unsigned int avrg = 0;

        for(i = 0; i < route_length; i++) {
                min = min(min, values[i]);
                avrg += values[i];
        }
        avrg = avrg / route_length;

        // Do the black magic.
        return (((IPT_BEEADHOC_MAXHOPS - (route_length-1)) * min *
            avrg) + 1) / 44;
}
```

Listing 13.15: forager_rate_lifetime

insert_forager_into_array

This is a helper function for `beeadhoc_forager_put`. Given a forager, its `opt_type` and its `struct daddr_list_entry` the function checks if the array used to store the foragers is already full. In this case the forager is freed and discarded.

Otherwise a rating is obtained and the forager is stored at the approriate location of the forager array of the `struct daddr_list_entry`. This location is calculated from the offset for the given `opt_type` and the number of foragers already stored there. Afterwards `count[]` and `countsum` are incremented. The timestamp for the `struct forager` is set to the current time.

```
static int insert_forager_into_array(struct forager* forager,
                                        struct daddr_list_entry *list
                                            ,
                                        __u8 opt_type)
{
        list->timestamp = get_seconds();
        /*
         * Check if array is full.
         */
        if(list->count[opt_type] == IPT_BEEADHOC_ARRAYSIZE) {
                beeadhoc_struct_free_forager(forager);
                return -1;
        }

        forager->nr_dances = forager_rate_lifetime(forager);

        list->forager[list->count[opt_type] + opt_type *
           IPT_BEEADHOC_ARRAYSIZE] = forager;
        list->count[opt_type]++;
        list->countsum++;
        forager->timestamp = list->timestamp;
        return 0;
}
```

Listing 13.16: insert_forager_into_array

get_opt_forager

This is a helper function for the API-call `beeadhoc_forager_get`. It is already called with the correct `daddr_list_entry`, the `opt_type` and a flag if the forager is to be removed from the array.

As all foragers are stored in an array in their respective `daddr_list_entry` getting a random forager is relatively simple: Generate a random number $0 < random < count[\texttt{opt\_type}]$, add an `opt_type`-related offset and read the forager from the array.

Now we calculate the age of the chosen forager. If it is not too old and we don't want to keep it (`IPT_BEEADHOC_FORAGER_KEEP`), we can just return it.

If we want to remove the forager more care must be taken. First we check if it is a candidate for dancing. If so, we copy it and return the copy.

Otherwise we decrement the members `count[]` and `countsum`, and copy the last forager in the array to the slot where we removed our random forager. The last slot is overridden with a `NULL`-pointer for easy error detection. This way the array is always filled from the start and selecting one at random remains simple. At last we check the foragers lifetime and discard it if it is already too old unless it is the last forager for this destination.

It might happen that we discard the last forager for a specific `opt_type`, but more foragers of different types are still available. In this case we return `NULL` and the calling function is responsible for getting a forager with another `opt_type`.

Again at the end of the function we update the timestamp of the `daddr_list_entry` to aid the garbage collection.

```
static struct forager* get_opt_forager(struct daddr_list_entry *list,
                                       __u8 opt_type,
                                       __u8 remove)
{
        static struct forager *forager;
        static unsigned int random;
        static unsigned int age;
        static unsigned int offset;

        offset = opt_type * IPT_BEEADHOC_ARRAYSIZE;
        do {
                /*
                 * I know modulo 'count' is not really random,
                 * but it's good enough for small 'count'
                 */
                random = (net_random() % list->count[opt_type]) +
                    offset;
                forager = list->forager[random];
                age = get_seconds() - forager->timestamp;
                if(remove == IPT_BEEADHOC_FORAGER_KEEP) {
                        if(age < IPT_BEEADHOC_FORAGER_LIFETIME)
                                break;
                } else {
                        if((forager->nr_dances > 0) &&
                                (age < IPT_BEEADHOC_FORAGER_DANCETIME
                                    )) {
                                forager->nr_dances--;
                                forager =
                                    beeadhoc_struct_copy_forager(
                                    forager);
                                break;
                        }
                }
                list->count[opt_type]--;
                list->countsum--;
                list->forager[random] = list->forager[list->count[
                    opt_type] + offset];
                list->forager[list->count[opt_type] + offset] = NULL;
                if(age < IPT_BEEADHOC_FORAGER_LIFETIME) {
                        /* We found him. Let's exit */
                        break;
                }
                if(list->countsum == 0) {
                        break;
                }
                beeadhoc_struct_free_forager(forager);

                if(list->count[opt_type] == 0) {
                        return NULL;
                }
        } while(1);

        list->timestamp = get_seconds();
        return forager;
}
```

Listing 13.17: get_opt_forager

13.4. ProcStat

In order to create a simple overview of internal data structures an output to the Proc Filesystem was written. The ProcFS is a virtual filesystem mounted onto the filesystem-tree. It is provided by the kernel itself - some of the entries only display information, some display settings and allow to change them, and others entries are able to write information to the kernel. A user can now view the internal BeeAdHoc data everytime. In order to view data just type `cat /proc/net/beeadhoc` or `cat/proc/net/beeadhoc_energy`. All code about ProcFS output is located in `ipt_BeeAdHoc_stat.c` and is implemented as seen below:

```

#include <linux/proc_fs.h>

#define procfs_name "net/beeadhoc"
 struct proc_dir_entry *Prof_Proc_File;
 int procfile_read_stat(char *buffer,
            char **buffer_location,
          off_t offset, int buffer_length, int *eof, void *data)
{
    int ret;
    if (offset > 0) {
        ret  = 0;
    } else {
        /* fill the buffer, return the buffer size */
    ret = sprintf(buffer, "*******************\n");
    ret  += sprintf(buffer + ret,"***BeeAdHoc Stat***\n");
    ret  += sprintf(buffer + ret,"*******************\n");
    }
    return ret;
}
void prof_init(void){ Prof_Proc_File =
create_proc_entry(procfs_name1, 0644, NULL);
    if (Prof_Proc_File == NULL) {
        remove_proc_entry(procfs_name1, &proc_root);
        printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
               procfs_name1);
    }
    Prof_Proc_File->read_proc = procfile_read_prof;
    Prof_Proc_File->owner     = THIS_MODULE;
    Prof_Proc_File->mode      = S_IFREG | S_IRUGO;
    Prof_Proc_File->uid       = 0;
    Prof_Proc_File->gid       = 0;
    Prof_Proc_File->size      = 37;
}
```

Listing 13.18: ipt_BeeAdHoc_stat.c

The output is only produced when the ProcFS file is read by 'cat' or any other viewer - so there is no influence on the overall performance in normal operation. A sample output looks like this:

```
************************************
************BeeAdHocStat*************
************************************
 received Scouts 90 (back at source 0, at dest 90)
 sent Scouts 90(timeout 0)
 forwarded Scouts 948(forth 553, back 395)
 received Swarms 0, send Swarms 0, forwarded Swarms 0
 received Forager 23518, sent Forager 16589, forwarded Forager 2698
 Foragerbytes_in 420888, Foragerbytes_out 378128,
 **********  scout seenlist **********
Src 20 :Dest 21 :ID 71
Src 20 :Dest 22 :ID 4
Src 20 :Dest 23 :ID 4
Src 20 :Dest 24 :ID 2
Src 30 :Dest 20 :ID 2
Src 22 :Dest 23 :ID 539
Src 24 :Dest 25 :ID 76

 ********* forager hashtable *********
:daddr age info nr_dances:hop1:hop2:...:hopn forager hashtable: end
6 forager (Age: 0) for daddr 25
 1 forager for opt_type 0
   0 (daddr 25) Age: 25  Info: fff Dances: 5 Hops: 29 28 30 25
 1 forager for opt_type 1
   0 (daddr 25) Age: 25  Info: fff Dances: 6 Hops: 29 28 30 25
 4 forager for opt_type 2
   0 (daddr 25) Age: 3  Info: ffffff Dances: 5 Hops: 29 28 30 25
   1 (daddr 25) Age: 1  Info: ffffff Dances: 6 Hops: 29 28 30 25
   2 (daddr 25) Age: 0  Info: ffffff Dances: 5 Hops: 29 28 30 25
   3 (daddr 25) Age: 1  Info: ffffff Dances: 5 Hops: 29 28 30 25
```

Listing 13.19: procfs output

In this listing one can get a detailed overview of scouts, foragers and swarms. We display not only the total number, but also forwarded, sent and received packets for each category. Furthermore the internal scout seenlist, which guarantees that no scout is handled twice by this node, is printed out here.

The last table shows foragers at this node sorted by their forager type. The `daddr` indicates the destination address of this forager (here only the last part of the IP is displayed). `Info` indicates collected information of this forager on its way. The `Hops` field indicates the route the forager will take.

Another ProcFS file states energy consumption. To read out energy the following code was used:

```
void startProfile_forager(void) {
    asm(
    "pushl  %eax \n\t"
    "pushl  %edx \n\t"
    );
    asm(
    "cpuid \n\t"
    "rdtsc \n\t"
    : "=A" (startCycle_forager)
    :
    );
    asm(
    "popl %edx \n\t"
    "popl %eax \n\t"
    );
}

unsigned long endingProfile_forager(void) {
    unsigned long endCycle_forager; //End cycle count
    asm(
    "pushl  %eax \n\t"
    "pushl  %edx \n\t"
    );
    asm(
    "cpuid \n\t"
    "rdtsc \n\t"
    : "=A" (endCycle_forager)
    :
    );
    asm(
    "popl %edx \n\t"
    "popl %eax \n\t"
    );
return endCycle_forager-startCycle_forager; }
```

Listing 13.20: cycle count

With the help of this code one can measure the cycle count used by the processor for processing an instruction or a block of instructions. The analysed instructions must be surrounded by startProfile_forager and endProfile_forager.

13.5. Scouting inside the kernel

The motivation behind the scouting subsystem in the BeeAdhoc kernel module is our aim to implement the routing algorithm complete the kernel space. In an earlier project group a hybrid model in the user space and kernel space part was used to implement a routing algorithm. But the communication between the two parts is difficult. One of the most important part of our kernel module is scouting which is responsisble for searching routes through the network and the handling of all scout packets.

Scout datastructs The scouting subsystem has three important datastructs. The first datastruct is the scout packets itself.

```
struct scout_packet {
        __u8 mode;
        __u32 id;
        __u8 ttl;
        __u8 nh;
        __u32 src;
        __u32 dest;
        __u32 hop[IPT_BEEADHOC_MAXHOPS];
        __u8 info[IPT_BEEADHOC_MAXHOPS];
};
```

Listing 13.21: scout packet

A scout can be in two different modes. If the mode value equals "1" then the scout is on the way to the destination and if it equals "2" then scout is on the way back to the source. The field id is a unique identifier for a scout of a certain destination. The scout packet can carry a route with "IPT_BEEADHOC_MAXHOPS" entries and the quality information for a hop.

The second important structure in our kernel module is the rxmt_entry in the retransmit array in which we save a scout packet after sending it. In this way we could implement "expanding ring search" (to be introduced shortly).

```
struct rxmt_entry {
        __u32 dest;
        struct scout_packet pack;
        struct timer_list timer;
};
```

Listing 13.22: retransmit entry

The retransmit entries is identified by a destination field. An important field of this structure is a timer_list which is relevant for scheduling a certain function at a certain moment to retransmit the scout packet. All retansmit entries are saved in an array.

The last important datastruct for the scouting subsystem is the "scout seen array" which saves the updated identifier for every destination and every source which visits a given node. This structure is used in functions and hooks needed for scout handling.

```
struct scoutarraypart{
        __u32 dest;
        __u32 src;
        __u32 id;
};
```

Listing 13.23: scout seen array entry

Route finding The scouting subsystems starts finding a route if a data packet from higher layers is stored in the bufferqueue because no forager with its destination is available on the dance floor.

Scout sending The sending of scout is realized with an UDP socket and a tasklet which is scheduled if a scout have to be sent. The destination port of a scout packet is 1124 (SCOUTPORT) .

```
static void scout_send_func(unsigned long data)
{
        static struct msghdr msg;
        static struct iovec iov;
        static struct sockaddr_in to;
        struct scout_speicher* sata = (struct scout_speicher *) data;

        to.sin_family = AF_INET;
        to.sin_addr.s_addr =  sdata->target;

        to.sin_port = htons( (unsigned short) SCOUTPORT);
        memset(&(to.sin_zero),'\0',8);
        msg.msg_name = &to;
        msg.msg_namelen = sizeof(to);
        iov.iov_base = &(sdata->scoutdata);
        iov.iov_len  = sizeof(sdata->scoutdata);

        msg.msg_control = NULL;
        msg.msg_controllen = 0;
        msg.msg_iov     = &iov;
        msg.msg_iovlen = 1;

        .
        .

        oldfs = get_fs();
        set_fs( KERNEL_DS );
        sock_sendmsg( clientsocket,&msg , iov.iov_len);
        set_fs( oldfs );
}
```

Listing 13.24: scout send tasklet

In the data field the destination address and the sent scout packet is saved. The destination address can be either next hop address or source address in the case of a scout flying back to the source or the broadcast address (255.255.255.255) in the case of a scout flying to a certain destination.
For finding a destination we also implemented a "expanding ring search". This is a standard technique in which at first a scout packet is sent with a small TTL and when this scout packet does not come back, a second scout packet with a decreased TTL is sent to the same destination. If a scout packets comes back the "expanding ring search" is over. For "expanding ring search" the retransmit entry is used to wait for a scout with a certain TLL and to start the retransmission. If a scout is sent, a retransmit entry is placed in the array with a copy of the scout packet and a timer.

```
        rxen->pack=scoutdatern.scoutdata;
        init_timer(&(rxen->timer));
        rxen->timer.function = ack_timeout;
        rxen->timer.expires = jiffies + SCOUTRESENDRATE;
        rxen->timer.data = (unsigned long) &(rxen->pack);
        add_timer(&(rxen->timer));
```

Listing 13.25: Copy scout packet and add timer

If the retransmit entry is placed in the array a timer is added which is scheduled at a certain moment and the function ack_timeout is executed. The function ack_timeout gets a pointer to the scout packet as a parameter and then the scout packet is sent with a new id and TTL and is stored in the retransmit entry.

Scout handling The scout handling is mostly done in the INPUT-Hook of the netfilter architecture. With the help of some structs and functions we use the hooks of the netfilter architecture to execute our code for every packet which passes the hook. For every incoming packet a function called isscout(..) checks if a packet is a scout packet which is specified a part value of 1124 in UDP header. If a scout packet is detected, then handling starts. As mentioned above a scout packet can be on the way to the destination which is shown by a mode value of ”1” or on the backward journey to the source which is mode ”2”.

```
static unsigned int scoutinhook(unsigned int hooknum,
                               struct sk_buff **pskb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn) (struct sk_buff *))
{




        .
        .
        unsigned char *headtemp = skb->data+sizeof(struct iphdr)+
            sizeof(struct udphdr);
        struct scout_packet *inscout =(struct scout_packet*) headtemp
            ;
        if (inscout->mode == 1){
                if (indev->ifa_list->ifa_local == inscout->dest){
                        .
                        .
                        tasklet_schedule (&send_scout_tasklet);
                        return NF_DROP;
                }else{
                        .
                        .
                        tasklet_schedule (&send_scout_tasklet);
                        return NF_DROP;
                }
        }else{
                if (inscout->mode == 2){
                        if (indev->ifa_list->ifa_local == inscout->
                            src){
                                .
                                .
                                makeforager(..);
                                rxmt_remove(..);
                                return NF_DROP;
                        }else{
                                .
                                .
                                tasklet_schedule (&send_scout_tasklet
                                    );
                                return NF_DROP;
                        }
                }
        }
        return NF_ACCEPT;
}
```

Listing 13.26: scout handling

The next step in the hook is a check if the scout packet is at the destination. If it is not then it will be checked if a scout packet with the same identifier had already enter the node. Otherwise the destination and the id are saved in the "scout seen array" and the scout packet are broadcasted again into the network. If the scout packet had already entered the node the packet would be dropped. If the packet is at the destination a scout with the mode 2 will unicasted back to the source. If a scout is back at the source, it recruits number of foragers depending on the quality of a route. Then retransmit entries are simply deleted.

Route rating and forager creation Currently a scout can rate the quality of its path based on the remaining battery level. Like forager handling at every mode a scout is filled with the informations of this node. At the destination the information is evaluated in the scoutinhook. A scout is filled with the information of a node in a similar as far a forager.

```
        __u32 info = 0;
        info = info | inscout->info[0];
        int i;
        for(i = 1; i < inscout->nh+1; i++) {
                info = info << 3;
                info = info | inscout->info[i];
        }
        info = info <<8;
        info = info >>8;
```

Listing 13.27: Information evaluating

After evaluating the information the retransmit entry is deleted and new foragers are created. Because the route is saved in a scout packet in an array it have to be copied into a source_route structure. Then the fields ”route”, ”daddr” and ”info” are filled with the information of the scouts. Then for every bee type several foragers are created depending upon the dance value

```
static void makeforager(....){
        int routebegin = 0;
        int i=0;
        struct list_head *route;

        route = kmalloc(sizeof(struct list_head),GFP_KERNEL);
        INIT_LIST_HEAD(route);
        routebegin=nh;
        if (routebegin>0) {
                for (i=0;i<routebegin;i++){
                        struct source_route *nextHop = kmalloc(sizeof
                            (struct source_route),GFP_KERNEL);
                        nextHop->hop = hop[i];
                        list_add_tail (&nextHop->list, route);
                }
        }
        __u32 startpoint = dest;
        struct source_route *srcHop=kmalloc(sizeof(struct
            source_route),GFP_KERNEL);
        srcHop->hop=startpoint;
        list_add_tail(&srcHop->list,route);

        struct forager *newForager = kmalloc(sizeof(struct forager),
            GFP_KERNEL);
        newForager->route = route;
        newForager->daddr = startpoint;
        newForager->timestamp = timestamp;
        newForager->info = info;
        newForager->nr_dances = nr_dances;
        __u8 opt_type;
        for(opt_type = 0; opt_type < IPT_BEEADHOC_OPT_MAX-1; opt_type
            ++) {
                struct forager *otherforager =
                    beeadhoc_struct_copy_forager(newForager);
                beeadhoc_forager_put(otherforager,opt_type,
                    IPT_FORAGER_OWN);
        }
        beeadhoc_forager_put(newForager, IPT_BEEADHOC_OPT_MAX - 1,
            IPT_FORAGER_OWN);

        LastForagerDest= newForager->daddr;
        tasklet_schedule(&queue_tasklet);
}
```

Listing 13.28: creating a forager

and the queue_tasklet is scheduled because that sends the waiting queue packets.

13.6. Iptables

Iptables BeeAdHoc-target

Writing the shared library for iptables was relatively an easy task. It does not need to check any parameters, so there is very little logic inside the code. Consequently, the code was copied from the `TARPIT`-target and modified to support the `BEEADHOC`-target. The code consists of mostly empty (required) functions and a struct with pointers to these.

Iptables IPV4OPTIONS-match

The shared library for the ipv4options match is included in the standard iptables distribution. We just had to modify it to be able to support iptables for our BeeAdHoc option. The usage stays the same, we just added `--beeadhoc` for matching beeadhoc packets and `! --beeadhoc` for matching other packets into the known command line options of this module.

Using iptables

We want our algorithm to process every outgoing IP packet on a specified interface in the BeeAdHoc network. Exceptions to this rule are packets directed to ourself (to the IP of our outgoing BeeAdHoc interface), packets to the broadcast address and packets to UDP port 1124 and 1125 (Scouts and Swarms). All other packets directed on the

```
# outgoing traffic
iptables -A OUTPUT -t mangle -d {IP} -j ACCEPT
iptables -A OUTPUT -t mangle -d {BCAST} -j ACCEPT
iptables -A OUTPUT -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A OUTPUT -t mangle -p udp --dport 1125 -j ACCEPT
iptables -A OUTPUT -t mangle -d {IP}/{NETMASK} -j BEEADHOC

# for incoming traffic
iptables -I INPUT -t mangle -m ipv4options --beeadhoc -j BEEADHOC
iptables -A INPUT -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A INPUT -t mangle -p udp --dport 1125 -j ACCEPT

# to forward packets
iptables -A FORWARD -t mangle -m ipv4options --beeadhoc -j BEEADHOC
iptables -A FORWARD -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A FORWARD -t mangle -p udp --dport 1125 -j ACCEPT
```

Figure 13.2.: Configuring netfilter for our needs

BeeAdHoc interface have to pass the BEEADHOC target. Reasons for these exceptions are obvious: a) A node need not ask the way to itself, b) processing broadcasted packets is of little value, c) packets on port 1124 are handled as scout packets, d) packets on port 1125 are handled as swarm packets. For c) and d) a verification for the specified packet is also done.

Only incoming packets carrying the BeeAdHoc option are interesting for our algorithm to build foragers from them. All others can be ignored by us.The same principle applies to the forwarded packets. The ones carrying the BeeAdHoc option have to be modified to insert the bee specific data for the host. The resulting configuration is shown in figure 13.2.

There is also the possibility to accept all packets for BeeAdHoc. However, a small modification to the iptables rules is neccessary. BeeAdHoc can handle both of these possibilities without further modification to the kernel module. The change of iptables rules are very simple(figure 13.3).

13.7. Swarms implementation

A swarm packet is only needed for a proper UDP communication, in which data packets are not acknowledged. As a result, the foragers cannot implicitly return to their source

```
# outgoing traffic
iptables -A OUTPUT -t mangle -d {IP} -j ACCEPT
iptables -A OUTPUT -t mangle -d {BCAST} -j ACCEPT
iptables -A OUTPUT -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A OUTPUT -t mangle -p udp --dport 1125 -j ACCEPT
iptables -A OUTPUT -t mangle -d {IP}/{NETMASK} -j BEEADHOC

iptables -I INPUT -t mangle -j BEEADHOC
iptables -A INPUT -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A INPUT -t mangle -p udp --dport 1125 -j ACCEPT

iptables -A FORWARD -t mangle -j BEEADHOC
iptables -A FORWARD -t mangle -p udp --dport 1124 -j ACCEPT
iptables -A FORWARD -t mangle -p udp --dport 1125 -j ACCEPT
```

Figure 13.3.: Configuring netfilter for our needs (without kernelpatch)

in acknowledgments. Because of this it is the job of a destination node to send these foragers back to the origin within a swarm packet. But the swarm capability is not only restricted to UDP. It is also possible that swarms will fly during a TCP communication. All changes for the swarms could be found in `.ipt_BEEADHOC_swarms.c`

13.7.1. Foreign and Own Forager

For this purpose we need to introduce a new concept of forager. The 'normal' forager can be categorized as 'own or 'foreign'. A 'foreign' forager is not created at a given node while an 'own' is created at a given node.We need to add another array of 'foreign' forager to each destination list entry:

```
struct daddr_list_entry {
        struct list_head list;

        __u32 daddr;
        int count[IPT_BEEADHOC_OPT_MAX];
        int countsum;
        unsigned long timestamp;
        struct forager* forager[IPT_BEEADHOC_ARRAYSIZE *
            IPT_BEEADHOC_OPT_MAX];

        int foreign_count[IPT_BEEADHOC_OPT_MAX];
        int foreign_countsum;
        unsigned long foreign_timestamp;
        struct forager* foreign_forager[IPT_BEEADHOC_ARRAYSIZE *
            IPT_BEEADHOC_OPT_MAX];
};
```

Listing 13.29: daddr_list_entry struct

Therefore every function in `ipt_BEEADHOC_struct.c` is to be adjusted. To get an overview of how the communication between two nodes A and B is done we will give an example: If a node A sends a forager to node B, node B will save this forager in it's 'foreign' forager list for destination A. The same step is repeated at node A if node B sends 'own' foragers to node A. So each node gets more and more 'foreign' foragers. Therefore we need a new type of a send queue. Each node takes it's 'own' foragers at first to send them to a specified destination. If 'own' foragers do not exist then it would send 'foreign' foragers. If the communication between node A and B is TCP based there would be no problem because each node will send it's 'own' foragers to the destination and the destination piggybacks them in the acknowledgements. However, in case of UDP traffic, the 'foreign' foragers are not piggypacked, therefore, a node explicitly sends its 'foreign' foragers to their source node through swarm packets.

13.7.2. Swarm triggering

A swarm packet will only be sent if there are not enough 'own' foragers. This means that the communication appears to be unidirectional. The other condition is that there must be more than a specified number of 'foreign' foragers in the array. Listing 13.30 shows the conditions in order to send a swarm.

```
if (
        (anzForager(startpoint,IPT_FORAGER_FOREIGN) >
            IPT_BEEADHOC_SWARMSIZE)&&
        (anzForager(startpoint,IPT_FORAGER_OWN) <
            IPT_BEEADHOC_SWARMSIZE_OWN)&&
        (iph->saddr!=in_aton('127.0.0.1'))&&
        (startpoint != index2->ifa_list->ifa_local)){
                BuildSwarm(startpoint,iph->daddr,bee_type);
                tasklet_schedule(&send_scout_tasklet);
        }
```

Listing 13.30: swarm send condition

13.7.3. Build the swarm packet

As you can see there is a function called `BuildSwarm(__u32 dest, __u32 src, __u8 bee_type)`. This function is responsible for building the proper swarm packet (Listing 13.31).

```
struct swarm_packet{
        __u32 src, dest;
        __u8 bee_type;
        __u8 pos;
        struct send_forager foragerpack[IPT_BEEADHOC_SWARMSIZE];
        __u32 hop[IPT_BEEADHOC_MAXHOPS];
}
```

Listing 13.31: swarm packet struct

The source, destination and bee type are saved in a swarm packet. Furthermore the swarm also needs routing information to get to it's destination (the position and hop variables are called pos and tos). The last information is the foragerpack that consists of a specified number of foragers. For this purpose we need a struct different from the normal forager because the normal forager is a hash table entry and we need an independent variable. So we defined a new struct so send_forager (Listing 13.32).

```
struct send_forager {
        __u32 route[IPT_BEEADHOC_MAXHOPS];
        __u32 daddr;
        __u32 route_length;
        unsigned long timestamp;
        __u32 info;
        __u8 nr_dances;
};
```

Listing 13.32: send_forager struct

The BuildSwarm function (Listing 13.33) can now build the swarm packet.

```
/* build swarm paket for given destination */
static int BuildSwarm(__u32 dest, __u32 src,__u8 beetype){
startProfile_swarm();
        scoutdatern.swarmdata.dest=dest;
        scoutdatern.swarmdata.src=src;
        scoutdatern.swarmdata.beetype=beetype;
        scoutdatern.port=SWARMPORT;
        scoutdatern.swarmdata.pos=0;
        int counter;
        for (counter=0;counter<IPT_BEEADHOC_SWARMSIZE; counter++){
                struct forager my_forager;
                struct forager* my2_forager;
                my2_forager=&my_forager;
                if (beeadhoc_forager_get(dest, beetype,
                    IPT_BEEADHOC_FORAGER_REMOVE,&my2_forager,
                    IPT_FORAGER_FOREIGN)==-1)
                        return -1;
                struct send_forager new_forager;
                new_forager.route_length =
                    beeadhoc_forager_reverse_route(my2_forager,&
                    scoutdatern.swarmdata,dest,src);
                int i;
                for (i=0; i<IPT_BEEADHOC_MAXHOPS; i++){
                        new_forager.route[i] = scoutdatern.swarmdata.
                            hop[i];
                }
                new_forager.daddr = src;
                new_forager.info = my2_forager->info;
                scoutdatern.swarmdata.foragerpack[counter]=
                    new_forager;
                beeadhoc_struct_free_forager(my2_forager);
        }
        beeadhoc_forager_get_route(beetype,&scoutdatern.swarmdata,
            dest);
        scoutdatern.target=scoutdatern.swarmdata.hop[0];
        DEBUGP3('BuildSwarm dest %u, src %u\n',NIPQUAD1(dest),
            NIPQUAD1(src));
        endProfile_swarm();
        return 0;
}
```

Listing 13.33: buildswarm function

Besides setting source, destination and bee type the function also calls `beeadhoc_forager_get` each time it packs a new forager into the swarm packet until `IPT_BEEADHOC_SWARMSIZE` has reached. After getting the forager it reverses its route. Yes, all forager transformation will be done at the node that sends the swarm packet. So the destination of the swarm only has to put the arriving foragers into the array.The `beeadhoc_forager_reverse` function will reverse the route of the forager as it will be needed at the creater node of the foragers. After that we pack the forager in the swarm array and free the forager at this node. The last thing that a BuildSwarm function does is to set the route for the swarm and its next hop.

13.7.4. Sending and forwarding the swarm

After the build function finishes successfully that swarm packet will be sent to the destination. If an an error occurs building a swarm packet (ie. forager dies during swarm building) then no swarm is sent.
The swarm will be handled like a scout packet during transportation. Therefore, we modified the `isscout` function to recognize a swarm. If this function detects a swarm it will return 1. Afterwards we need to identify if it is a valid swarm packet and if the swarm packet arrived at it's destination or not (Listing 13.34).

```
if (inhooktemp == 1){
        unsigned char *headtemp = skb->data+sizeof(struct iphdr)+
            sizeof(struct udphdr);
        struct swarm_packet *inswarm =(struct swarm_packet*) headtemp
            ;
        DEBUGP3('scoutinhook: - Swarm Paket empfangen.\n');
        if (indev->ifa_list->ifa_local == inswarm->dest){
                DEBUGP3('Swarm am Ziel\n');
                put_swarm(inswarm->beetype,inswarm);
                LastForagerDest= inswarm->dest;
                tasklet_schedule(&queue_tasklet);
                return NF_DROP;
        }else{
                DEBUGP3('Swarm nicht am Ziel, weiterleiten\n');
                ResendSwarm(inswarm);
                tasklet_schedule (&send_scout_tasklet);
                return NF_DROP;
        }
}
```

Listing 13.34: swarm arrival

If the local ip is the destination ip then the swarm arrived at its creater node and will be put into the array using the `put_swarm` function. Otherwise the swarm is sent to the next hop using the `ResendSwarm` function. No additional information about the `ResendSwarm` function is needed because it only has to set the next hop and then forward it.

13.7.5. Swarm arrives at destination

As shown above the arrival of a swarm calls the `put_swarm` function (Listing 13.35).

```
static void put_swarm(int opt_type, struct swarm_packet* swarmpack){
        int i;
        for(i = 0; i < IPT_BEEADHOC_SWARMSIZE-1; i++) {
                struct forager *newForager = kmalloc(sizeof(struct
                    forager),GFP_ATOMIC);
                static struct list_head *head;
                static struct source_route *temproute;
                head = kmalloc(sizeof(struct list_head), GFP_ATOMIC);
                INIT_LIST_HEAD(head);
                int counter;
                counter=0;
                for (counter=0;counter<=swarmpack->foragerpack[i].
                    route_length;counter++){
                        temproute=kmalloc(sizeof(struct source_route)
                            ,GFP_ATOMIC);
                        if (swarmpack->foragerpack[i].route[counter
                            ]!=0){
                                temproute->hop = swarmpack->
                                    foragerpack[i].route[counter];
                                list_add_tail((struct list_head*)
                                    temproute,head);
                        }
                }
                newForager->route = head;
                newForager->daddr = swarmpack->src;
                newForager->info  = swarmpack->foragerpack[i].info;
                __u8 bee_type = 2;
                beeadhoc_forager_put(newForager,bee_type,
                    IPT_FORAGER_OWN);
        }
}
```

Listing 13.35: put forager from swarms

This is a simply but important function. An incoming swarm will create a new forager for every (send_)forager in its payload. The route is copied into a new list and the forager is directly put into the array. Therefore, we have to ensure that valid foragers are encapsulated into the payload.

14. Performance Evaluation Framework for MANETs in Linux

14.1. Introduction

The 'performance evaluation framework' is made to test the capabilities of routing protocols in reality, (quasi-)reality (artificial limitation to the availibilty of the wireless devices) and in the UMLs (User Mode Linux). Our initial work focused on providing a common base for all platforms. This is described in the section 'scripts'.
Then we had to decide which tools are necessary for the framework. On the one hand we use real applications like 'FTP server and client' and on the other hand we use synthetic traffic generating tools. The real applications are good for showing that the routing protocol can be used in a real MANET. But for these applications it is difficult to measure parameters like 'packet delay', 'throughput', 'jitter' and 'packet delivery ratio'. But now we are able to measure 'throughput',('packet delay' - see below), 'packet delivery ratio', 'session delay' and 'session completion ratio' for real applications.
If we want to measure other parameters we have to use 'trafficgenerators'. In short these are programs which send TCP and/or UDP packets and measure important parameters like 'throughput', 'packet delay' etc. In the appendix of this report we describe two of these trafficgenerators.

14.2. Tested Algorithms

We want to compare BeeAdHoc with existing state-of-the-art algorithms like AODV and OLSR.

14.2.1. AODV

The algortihms was described in section 6.6.6 We choose an implementation of Uppsala University of Sweden [see [Norb]]. This implementation is implemented in a hybrid fashion between kernel and user space. In the kernel space the netfilter is used for filtering and enqueueing packets for user space. In the user space the main parts are implemented. It means that routing packets are handled here (route discovery, route maintaince, etc.)

For starting the AODV one has to compile the source code. After this one gets a kernel module and a user space daemon. If a netfilter packet is not compiled as a module, one has to comment the request depending upon availability of netfilter module.

To start the AODV one has to only start user daemon which loads the AODV module. Now one can run the tests.

14.2.2. OLSR

Link State Routing Protocol This type of routing protocol requires each router to maintain at least a partial map of the network. When a packet has to be sent the router can calculate the shortest(or the best) path. When a network link changes its state (up to down), a notification, called a link state advertisement (LSA) is flooded throughout the network. It means the message are sent to all neighbors. All the routers note the change, and recompute their routes accordingly.

The Optimisation OLSR is a proactive routing protocol for mobile ad hoc networks. The protocol inherits the stability of a link state algorithm and has the advantage of having routes immediately available when needed due to its proactive nature. OLSR is an optimization over the classical link state protocol, tailored for mobile ad hoc networks. OLSR minimizes the overhead from flooding of control traffic by using only selected nodes, called MPRs, that can only retransmit control messages. This technique significantly reduces the number of retransmissions required to flood a message to all nodes in a network. The protocol is particularly suited for large and dense networks, as the optimization done using MPRs works well in this context. The larger and more dense a network, the more optimization can be achieved as compared to the classic link state algorithm.

OLSR does not require any changes to the format of IP packets. Thus any existing IP stack can be used as is: the protocol only interacts with routing table management.

For more information see [CJ].

We take an implementation which is based on INRIA OLSR. The source code and documentation is available online at www.olsr.org. This implementation runs only in user space that is why it does not depend on the kernel version. To run OLSR one has to compile the source code and to start the user space daemon.

14.3. Testing environment and settings

14.3.1. Hardware

For our tests we had a number of laptops which were made available by LS III. For development, simple quasireality testing and UML testing we used our five Maxdata notebooks. Later any final tests were performed on twelve identical HP notebooks. So for final tests we used exactly the same test equipment to avoid any influence of hardware compability problems with the test results.

Hardware equipment in detail:

- Maxdata Pro 7100x

  Intel Pentium 4 M, 2,2 GHz, 512MB RAM, 40GB HDD, internal Wistron NeWeb 802.11b Wireless LAN PCI Card (5 Notebooks total)

- HP Omnibook x3e gc

  Intel Pentium 3, 1 GHz, 256MB RAM, 20GB HDD, Elsa AirLancer MC-11 PCM-CIA wireless LAN card (12 Notebooks total)

node	IP	MAC	Type
brecht	192.168.1.20	00022D498175	HP Omnibook x3e gc
kafka	192.168.1.21	00022D498183	HP Omnibook x3e gc
mann	192.168.1.22	00022D3F87A5	HP Omnibook x3e gc
lessing	192.168.1.23	00022D497FDF	HP Omnibook x3e gc
rilke	192.168.1.24	00022D497FE9	HP Omnibook x3e gc
heine	192.168.1.25	00022D497FEA	HP Omnibook x3e gc
hesse	192.168.1.26	00022D497FFD	HP Omnibook x3e gc
hauff	192.168.1.27	00022D497FF5	HP Omnibook x3e gc
schiller	192.168.1.28	00022D4007B3	HP Omnibook x3e gc
kleist	192.168.1.29	00022D4007D5	HP Omnibook x3e gc
goethe	192.168.1.30	00022D4007B4	HP Omnibook x3e gc
storm	192.168.1.31	00022D4007CC	HP Omnibook x3e gc
boell	192.168.1.32	000124D05B8E	Maxdata Pro 7100x
grass	192.168.1.33	000124D05DC5	Maxdata Pro 7100x
fontane	192.168.1.34	000124D068D7	Maxdata Pro 7100x

Table 14.1.: Detailed node overview

The UML test were done on a Maxdata notebook since these were the most powerful notebooks. The quasireality tests were done in one room - every notebook had the same distance to its neighbor. Any other interfering WLAN was disconnected and the onboard ethernet-cards were deactivated for every test.

14.3.2. Software

All tests were performed on a clean Debian Sarge Netinstall using Kernel 2.6.7 (see section 14.3.2). For our UML test we used a Debian system with kernel 2.6.7 patched with SKAS as a host system. The virtual machines ran a small Linux with Kernel 2.6.9. For connecting the machines we used a modified uml_switch (see A.1.4)

The following settings and parameters were used during our tests for the protocols:

- OLSR 0.4.9

  `olsrd -d 0 -i eth1`

- AODV 0.8.1

  `aodvd -l -r 3 -i eth1 -D -d -L`

- BeeAdHoc

  ```
  IPT_BEEADHOC_FORAGER_LIFETIME 2
  IPT_BEEADHOC_FORAGER_DANCETIME 2
  IPT_BEEADHOC_MAX_DANCES 8
  SCOUTLISTLENGTH 30
  RXMTLISTLENGTH 30
  SCOUTRESENDRATE 500
  IPT_BEEADHOC_SWARMSIZE 20
  IPT_BEEADHOC_SWARMSIZE_OWN 11
  ```

The following settings and parameters were used during our tests for the traffic generators (destination IPs and durations may differ from used parameters):

- thrulay 0.6

  `thrulay -t 900 192.168.2.20 > /logthrulay`

- sqtg

  ```
  tg -t 00:00:00 -p 0.01 -s 1024 -d 192.168.2.20 -e 900 -m 2.16
  -S 213000
  ```

- netperf 2.4.0

  `netperf -l 900 -H 192.168.2.20`

- D-ITG 2.4

  `ITGSend -l /logitg -T TCP -m rrtm -t 900000 -a 192.168.2.20`

Minimal Linux Installation

We got the motivation for minimal Linux installation from a project called APE. To guarantee that other programs and processes do not influence the communication process of BeeAdHoc or other MANET protocols we ensured that there are few or no such processes running on the machines. So we went to work to purify the installed Linux. We started by choosing the net-base-installation that is a small-sized installation. We decided to use the Sarge Debian Linux with a 2.6.7 kernel. There are no additional packages in this kind of installation. Because we had to compile the kernel and run user mode Linux machines we had to install more packages like: xfree, mc, traceroute, iproute, libc, kernel headers and so on. But to avoid influence on tests we configured all additional packages not as daemons and there are not started at boot up. Every bootup has minimal configuration but if needed the components like xwindow could be started. We did notnot install daemons or other runtime packages beside the existing ones. In addition we made sure that no needless cron job will be executed. At this point we had a minimal Linux installation and so we could concentrate on the tests and could be sure that there was no influence by other running programs.

14.4. Scripts

In the case of routing algorithms we have to emulate the reality and the complete enviroment in a simulator, like 'OPNET' and 'NS-2'. But these are only models of reality and environment. Because we implemented a routing algorithm in Linux, we wanted to test the algorithms on a real operating system and not only in a simulator. The next step is to test the algorithms in user-mode-linux which is a virtual linux running in the userspace. With the help of a switch one can communicate between several UMLs. For first tests we used this test environment, because it behaves almost like real nodes but without restrictions on bandwidth resources.
Testing with wireless LAN cards in reality is very difficult because of unpredicted topology changes and therefore non-reproducible results. So we made one modification in our testing organisation. We took normal notebooks with normal WLAN cards, but simulated the mobility by software - we call this quasireality. For the last tests it is interesting how the protcols behave in reality, but since it is hard here to create reproducible test results this data should not be used for a scientific comparison. Assume we have real notebooks with wireless LAN cards which are in a topology like in figure 14.1.

Because it is very difficult to predict the availability of the participants, it is easier to take a topology like in figure 14.2 and to create 14.1 by software blocking. First attempts

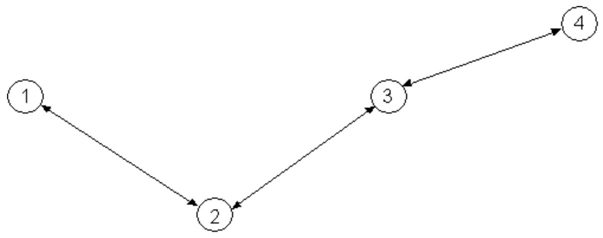


Figure 14.1.: real topology

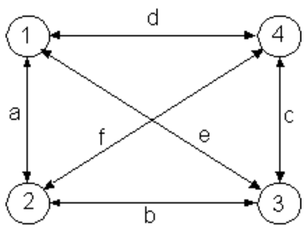


Figure 14.2.: topology: everybody sees everybody

were made to block traffic in the iptables prerouting table. In our tests we discovered that a) some protocols cannot be stopped to communicate by this and b) that under high loads the iptables let packets slip through the blockade. So we used the mackill [see A.12] kernel module from the APE testbed and adapted it to the linux kernel 2.6 branch.

In order to restore the topology in figure 14.1 one must cut connections d,e and f in topology from figure 14.1. This can be done by:

- node 1 `lockmac a 3 lockmac a 4`
- node 2 `lockmac a 4`
- node 3 `lockmac a 1`
- node 4 `lockmac a 1 lockmac a 2`

14.4.1. Motivation

Simulators like OMNeT++, ns2 or OPNET can collect a number of different measurement results in the simulation. Changing type of data transmission, shape of generated traffic and mobility of nodes is done in a user friendly GUI. In simulators like OMNeT++ one can choose e.g. between a session oriented and a non-session oriented data transmission. In bigger simulators like OPNET one can choose e.g. between TCP/IP and UDP/IP and e.g. FTP on the application layer, so one can simulate nearly the reality. Furthermore simulators have the possibility to use a scenario file to control the environment of the test. This file defines how nodes move and what type of data transmission is used. Our challenge was to build a 'performance evaluation framework' which should allow us to combine the collection of test data. In addition it should be flexible about parameters like mobility and data transmission types. One goal was to conduct tests in quasi-reality and user-mode-linux with the same scripts and parameters, so that the results are comparable. At this moment we can measure the parameters 'throughput', 'packet delay' , 'session delay', 'session completion ratio', 'jitter', 'total generated packets', delay standard deviation', 'control packets' and the number of 'total sessions' - depending on the used traffic generator. To generate different topologies for mobility testing we use our scenario editor (see A.9.3).

14.4.2. Design of the testing framework

The testing framework consists of a number of small scripts and programs which are controlled by one main tcl script. Detailed descriptions of these programs can be found in the appendix A of this report. Tests are started on all nodes by 'start x' where x

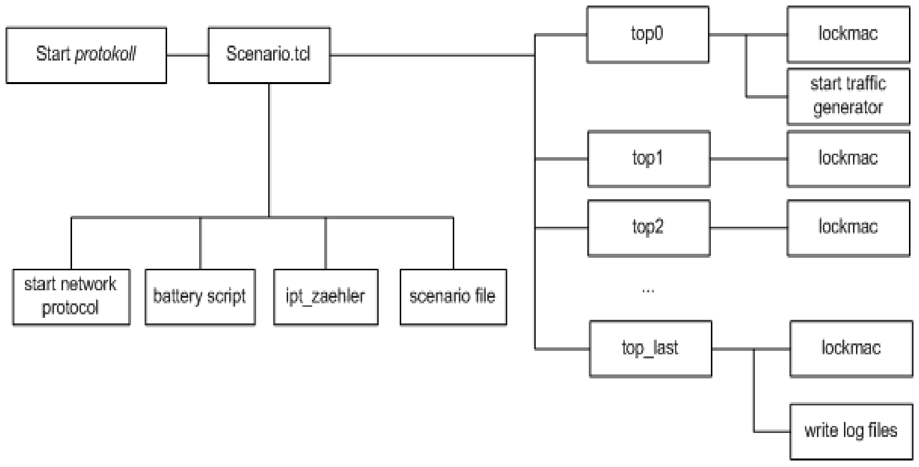


Figure 14.3.: Performance Evaluation Framework

is network protocol to be tested (in our case these are currently BeeAdHoc, OLSR and AODV). Once started on all nodes the startsync program awaits a keystroke from any node, so that all nodes start their scripts simultaneously.

The structure of the test-framework is shown in figure 14.3. The individual components are described below:

- `scenario.tcl`

  The main script is responsible for the following actions:

  - clear all existing mac blockings
  - unload any network protocol
  - synchronize nodes via startsync
  - get hostname for further actions
  - load appropriate network protocol
  - changing topology

  To change topology we use a small tcl script:

```
#!/usr/bin/tclsh set DIRECTORY scenarios/ set WLANCONF wlan.conf
    set
SCENARIO ${DIRECTORY}scenario

if {[catch {set f [open $SCENARIO]}] == 0} {
    puts "save old $WLANCONF"
    catch {exec cp $WLANCONF $WLANCONF.bak}
    while {[gets $f line] >= 0} {
    if {$line == ""} continue
    set line [string trim $line]
    if {[regexp {^([0-9]+) +([0-9]+)$} $line x s t] == 1} {
        if {[catch {exec cp $DIRECTORY${WLANCONF}_$s $WLANCONF}]
            == 0} {
        puts "use $DIRECTORY${WLANCONF}_$s for ${t}s"
        after [expr $t * 1000]
        } else {
        puts "cannot read $DIRECTORY${WLANCONF}_$s"
        }
    }
    }
    puts "szenarios finished"
    puts "restore old $WLANCONF"
    catch {exec mv $WLANCONF.bak $WLANCONF}
    catch {close $f}
} else {
    puts "cannot read $SCENARIO"
}
```

Listing 14.1: topologychange script

- `scenario file`
  The scenario.tcl script gets a scenario file that describes when to change to what topology. It might look like this:

  ```
  1 60
  2 20
  3 60
  ```

  Listing 14.2: sample scenario file

  In this example Topology 1 will last 60 seconds and the second topology will only last 20 seconds. The advantage of this script is the possibility to create bigger scenarios with a small number of topology files. Simply append the same topology file at the end of the scenario.

- `traffic generators`
  For our TCP and UDP tests we used a number of different types of traffic generators. For TCP we used thrulay (see A.2), D-ITG(see A.10) and netperf (seeA.3) - for UDP we used sqtg(see A.11) and D-ITG. Parameters are described in section 14.3.2.

- `startsync`
  Changing topology is very time-critical. In order to produce repeatable results all scripts must be started at exactly the same time. At this point we take the functionality from the 'APE testbed'[[NLTG]] to synchronize the nodes. The program 'startsync' is used to begin the test when all test participant are ready for the test. It waits until any key at any computer is pressed. This event is broadcasted in a packet, so that the test can start synchronously on all nodes.

- `lockmac`
  This simple script acts as a front-end to mackill (see A.12). Since mackill can only block nodes by their mac-address we developed lockmac for a simple conversion between mac-addresses and host names. Three different parameters are supported - 'a hostname' to block a node, 'd hostname' to deblock a node and 'clear' to delete all blockings.

- `topology files`
  These files describe the current arrangement of nodes in a network. Since in quasi-reality all nodes are located in one room the file simply states which node has contact to which nodes. Every network node has its own section (script must be started with parameter 'hostname') where its connectivity is declared through the lockmac script. Each script is a simple bash script which looks like this:

```
#!/bin/bash echo Topology 1 Node $1 case "$1" in

alpha)
        lockmac a gamma
        lockmac d beta
;; beta)
        lockmac d alpha
        lockmac d gamma

;; gamma)
        lockmac a alpha
        lockmac d beta

;; *)
    echo ende
;; esac
```

Listing 14.3: sample topology

Since changes for all nodes are declared in one file, we can simply copy the set of topology files to the other nodes in the network without the need to adjust them. The script also accepts normal shell commands and so allows us not only to block MAC adresses, but also to start our traffic generators and to write log files at the end of the scenario.

- `ipt_zaehler`
  Every network protocoll uses some kind of control packets. In order to count them a special kernel module was developed. It counts packets on the corresponding controlpacket port of the protocols.

- `battery script`
  This script is started in the background and catches the updated state of the battery in the 'ProcFS' (/proc/acpi/battery/BAT0/state) and writes this value for a later evaluation into a log file.
  This script one can execute with the command

```
echo "Zeit:$(date␣+%s.%N)␣␣␣mAh:$(grep␣remaining␣/proc/acpi/
battery/BAT0/*␣|␣awk␣'{printf␣$3;}')" >> ./Log/batlog
```

Listing 14.4: batlog script

every second and writes the result to a log file.

- `ProcStat`
  For BeeAdHoc we also dump the content of the ProcStat files(see 13.4) after the scenario to get in-depth information about the performance of our protocol.

Our goal is to measure at least the following parameters:

- Throughput
  How much data is transmitted in a certain time?
- Delay
  How long a packet take from its source to its destination?
- Packet delivery ratio
  How many packets reach at its destination host?
- Route discovery latency
  How long it takes to discover a route?
- Session completion ratio
  How many session are completed?
- Session delay
  How long a session takes?

14.4.3. APE

A good start for a testing framework was the Ad hoc Protocol Evaluation testbed (APE) framework from Uppsala University[NLTG] which was used to test their AODV routing protocol [Norb]

APE runs completely independent from a normal Linux or Windows installation. After installation a new entry in the boot manager is added to load APE. APE completely resides in an image file which is used for all file operation, so the data on the main HDD remains untouched. The system is based on a small Linux which allows testing without being interfered by any unnecessary background processes.

After starting the testbed the main menu appears in which the protocol and scenario can be chosen. APE then loads the appropriate driver and modules for WLAN cards and routing protocols. To guarantee valid and reproducible testing results a synchronization tool is started to sync clocks of the connected testbed systems. The test scripts have a simple structure - each machine has an identical script which is executed line by line. In this file there is a section for each connected system - these sections consist of command blocks which declare duration and commands. By declaring the duration it is assured that every command is executed only for the predetermined time. A typical example looks like this:

```
node.3.action.2.command=ping_node 0 2900 0.1 512
node.3.action.2.duration=40
```

In the command section every Linux command can be executed. At the end log files are saved containing all interesting values. A toolkit allows graphical evaluation of these results.

Unfortunately the current APE implementation is not able to return reliable results since it is a little buggy. The framework makes use of the sleep command which interferes with the communication tools. A FTP transfer can fail because the sleep command blocks feedback to the other hosts. The coarse scripting structure for our testing was adopted from here since it allows easy manipulation of the scripts without any special tools. Nevertheless a scripteditor was developed which allows us to create scenario files with a GUI.

Since APE was developed on a 2.4 kernel version and many tools only work with this version we could not use it nor adapt it. Another tool is mackill which is used to change topology by blocking packets from a specific mac-address. This tool does only run under kernel 2.4 so we developed our own procedure.

14.4.4. Cloning

But we have to be sure that the results and tests are not influenced by circumstances like missconfiguration or too few packets. To achieve this we have to think for some ideas. After all the first tests point out that we have to make this step necessarily (together with a very small Linux installation). We came to this conclusion as we stumbled on problems where we could not exclude configuration issues.
The idea was to make a fresh clean Linux installation that is very minimal. Afterwards we distributed this minimal installation to every laptop. That is a easy step because all laptops has the same hardware configuration. So it is guaranteed that every laptop have the same programs, data and configuration. Thus the requirements for the test were established and we could be certain that the problem is not a false configuration of one laptop.

14.5. Scenarios

During our test we used a variety of topology sequences. Because of the limited performance of the UML simulation we only used up to five nodes in our tests. For quasireality we used up to twelve nodes.

14.5.1. UML

Our first test ran inside a virtual network of five User-mode-linux nodes. All tests in section 14.6.5 are done with the following topologies(see Figures 14.4 and 14.5). The nodes `rilke`, `kafke`, `heine`, `goethe` and `lessing` undergo relative moderate topology changes. Tests here always include a server on `lessing` and a client application on `rilke` which demands data from the server.

For the tests these topologies were changed at different speeds. First a test was run with no change at all. A chain of nodes was connected in a row - a server at `lessing` and a client at `rilke`. Subsequently tests were run with a topology change every 5, 10, 30 and 60 seconds.

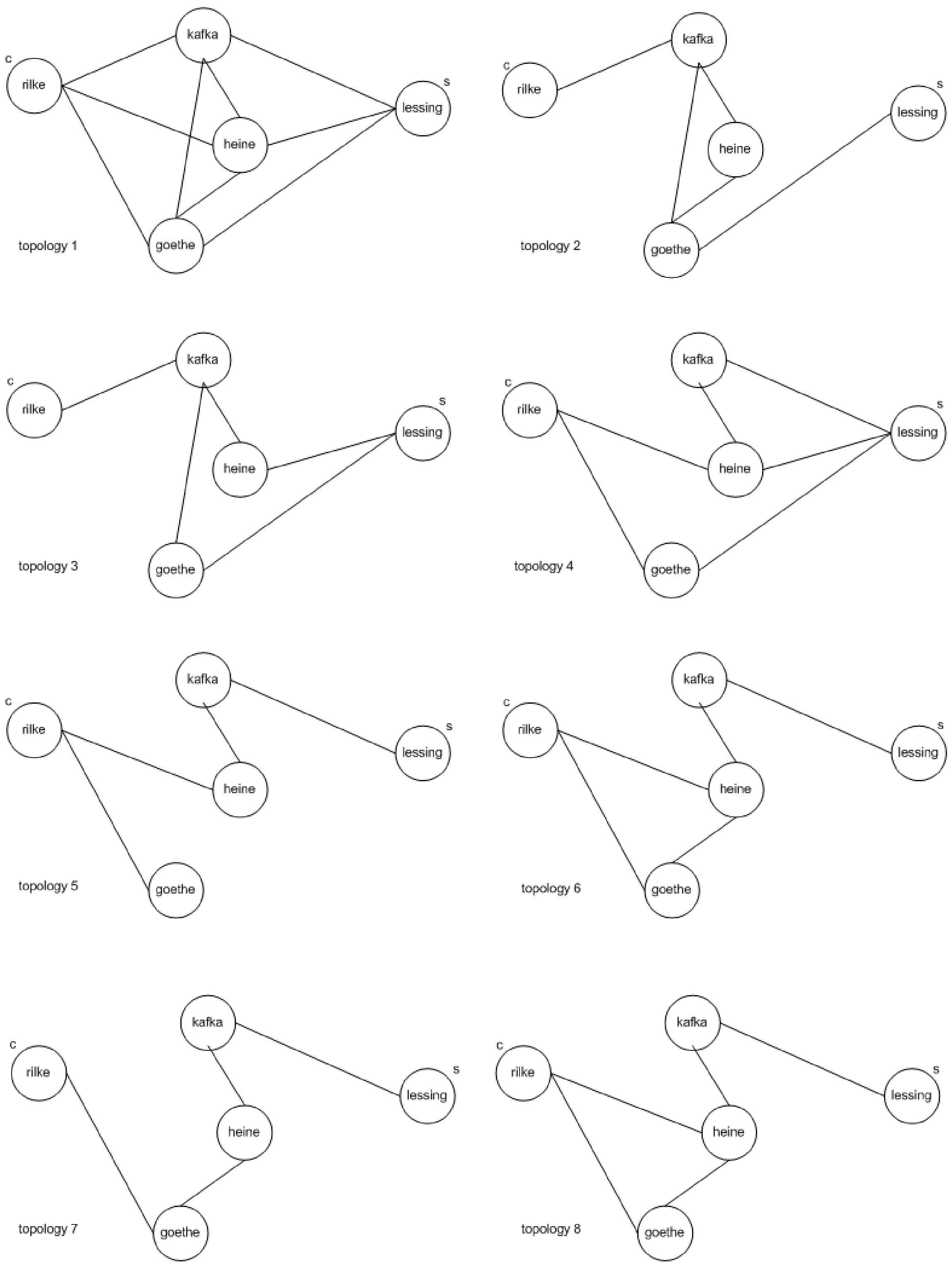


Figure 14.4.: UML topologies (part1)

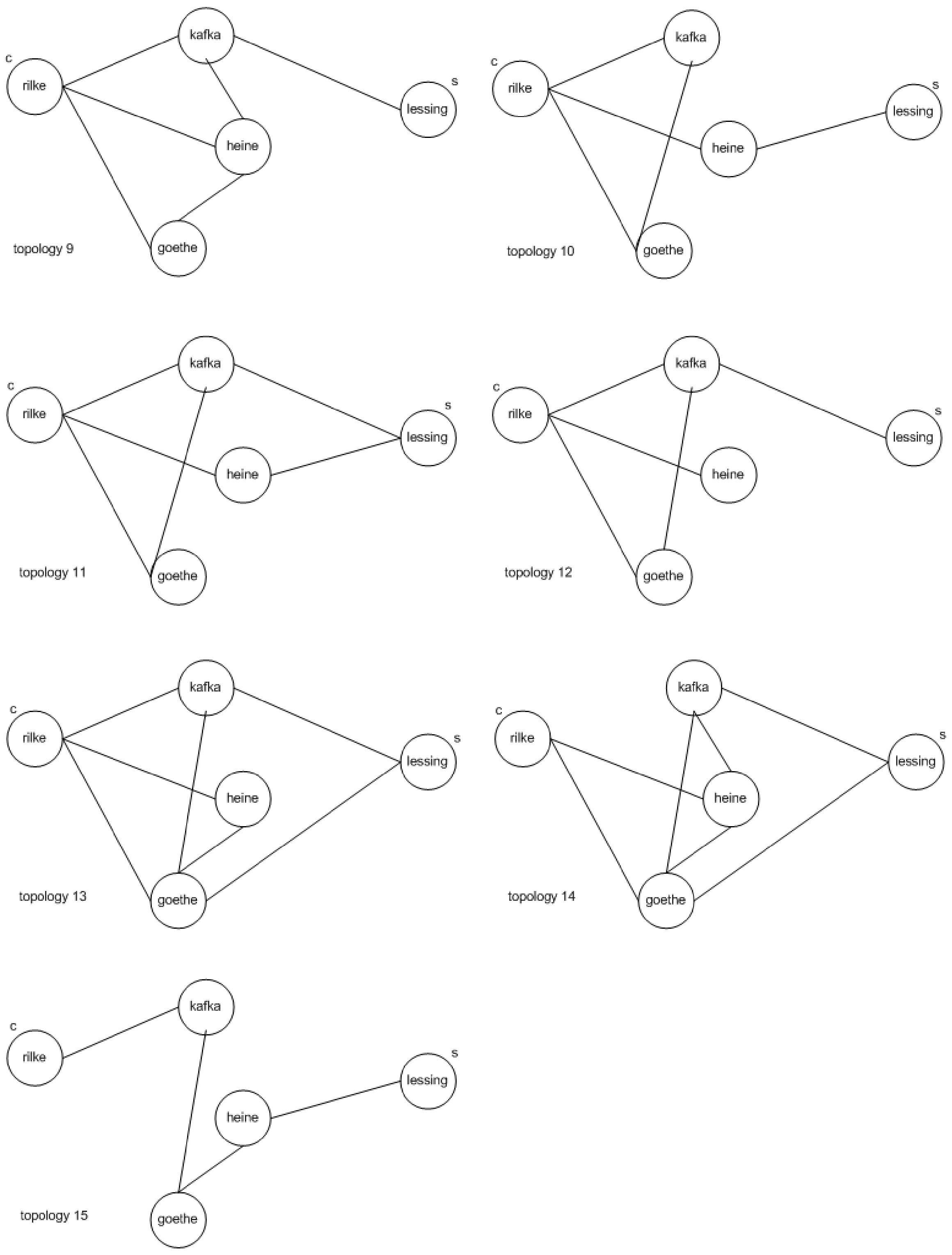


Figure 14.5.: UML topologies (part2)

14.5.2. Quasireality

Because of the availability of twelve nodes the topologies in quasireality are more complex and are also more challenging for the candidate algorithms. Some abrupt and extreme changes in the connectivity are sometimes difficult to manage for the protocols. Additionally we do not only test with one client/server pair but with three. In these scenarios `brecht`, `mann` and `rilke` are clients and `kafka`, `lessing` and `heine` their corresponding servers. Especially the connection between `mann` and `lessing` seemed to be very hard to hold up to and the performance of protocols significantly degraded in this case.

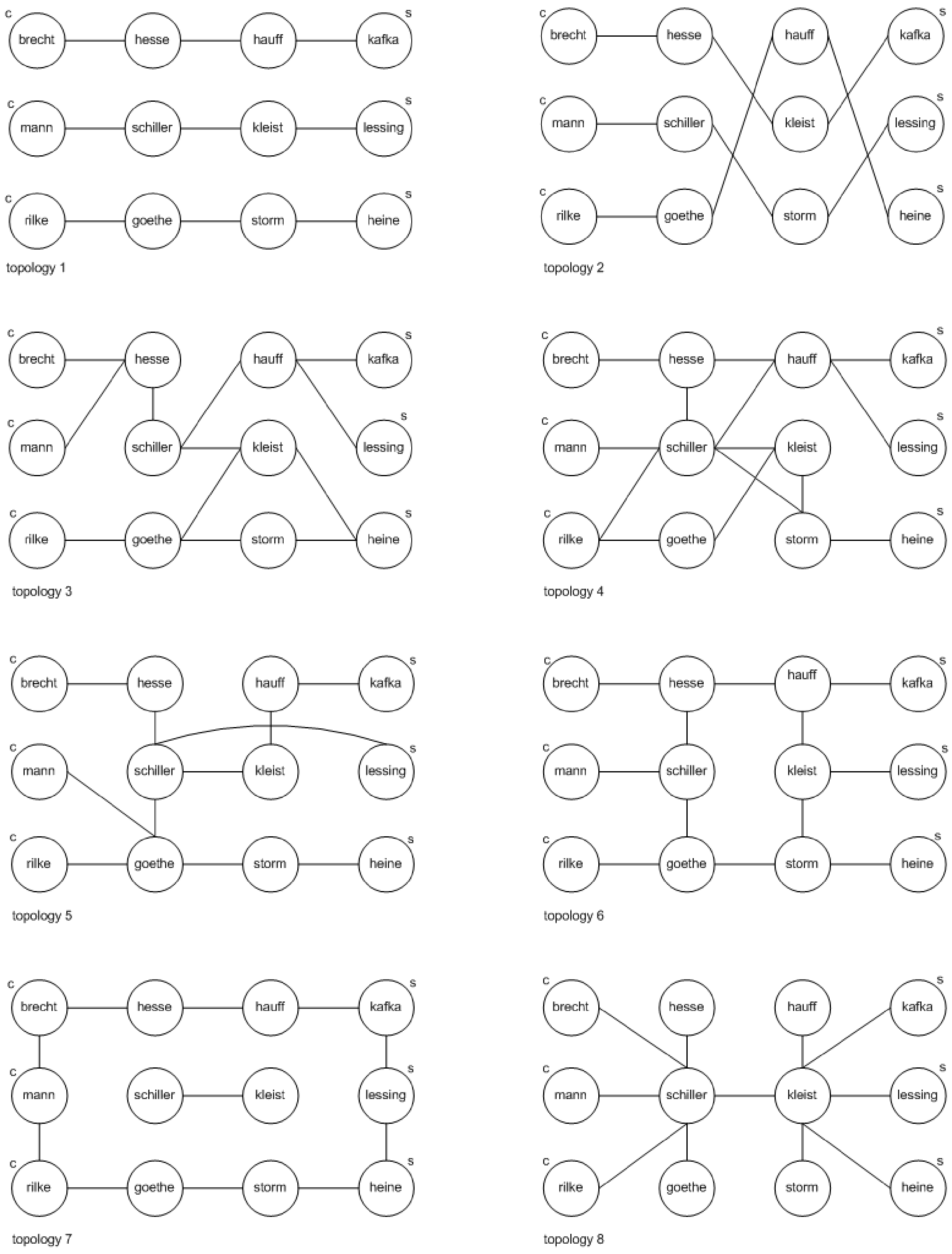


Figure 14.6.: quasireality topologies (part1)

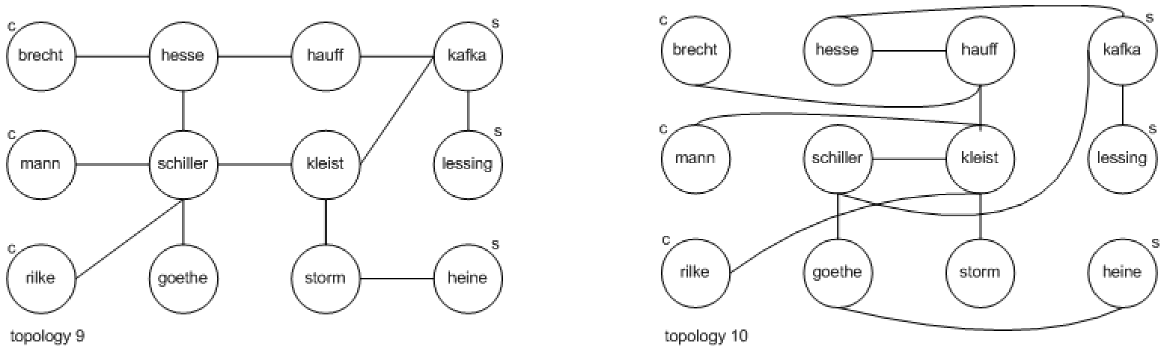


Figure 14.7.: quasireality topologies (part2)

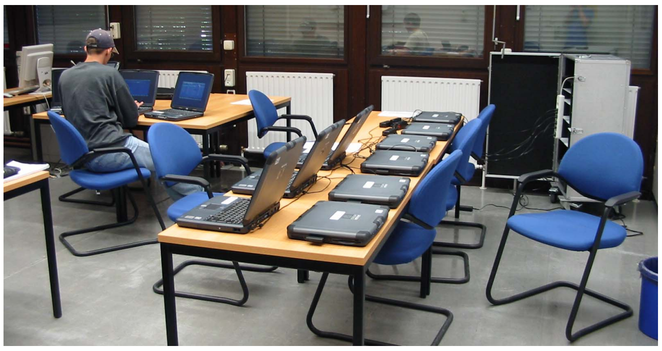


Figure 14.8.: quasireality test setup

14.5.3. Reality

Since testing in reality is quite unreproducible and unpredictable the results must be seen as a 'proof of concept'. Problems during these tests was the imprecise chance to predict connectivity caused by terrain topology (trees and buildings) and the persons on the testfield which influenced the connections. But despite all this we used this scenario because is reflects a real-life scenario where the algorithms have to operate hard to establish and hold connections.

For our tests we used our university campus area which is shown at figure 14.9. For the tests we used eight notebooks and performed six tests on them. Three tests using a static topology and three tests with mobile nodes (for each protocol one test). The complete test area measures about 1.6 hectare. The nodes traveled with speeds from 0 up to 5 km/h.

All tests run per 720 seconds. Tests began at Point F where all nodes were synchronized - after that each node moves to its stating point within 30 seconds. One thrulay session was performed from 'Goethe'(server) to 'Lessing' (client).

- `route C to B`
  Node Goethe travels with 4 km/h between point C and B.
- `route F to E`
  Node Kafka travels with 2,7 km/h between point F and E.
- `route A to F`
  Node Schiller travels with 2,5 km/h between point A and F.
- `route D to E`
  Node Rilke travels with xxx km/h between point D and E .
- `route A to B`
  Node Brecht travels with 2,5 km/h between point B and D.
- `route F`
  Node Storm is static and did not move during the test.
- `circle fast`
  Node Heine travels with 4,7 km/h from G to G
- `circle slow`
  Node Lessing travels with 3,1 km/h from H to H
-

For static tests the nodes are located as described in table refplace

Point	Node	Lon	Lat	Altitude
A	Goethe	7,4175350°	51,4929533°	113,6m
B	Brecht	7,4153800°	51,4928950°	117,3m
C	Kafka	7,4163483°	51,4929747°	116,9m
D	Schiller	7,4175267°	51,4932967°	116,2m
E	Rilke	7,4173950°	51,4940217°	116,7m
F	Lessing	7,4166200°	51,4934850°	116,6m
G	Heine	7,4167733°	51,4931217°	116,6m
H	Kleist	7,4175800°	51,4938200°	116,3m

Table 14.2.: static node placement

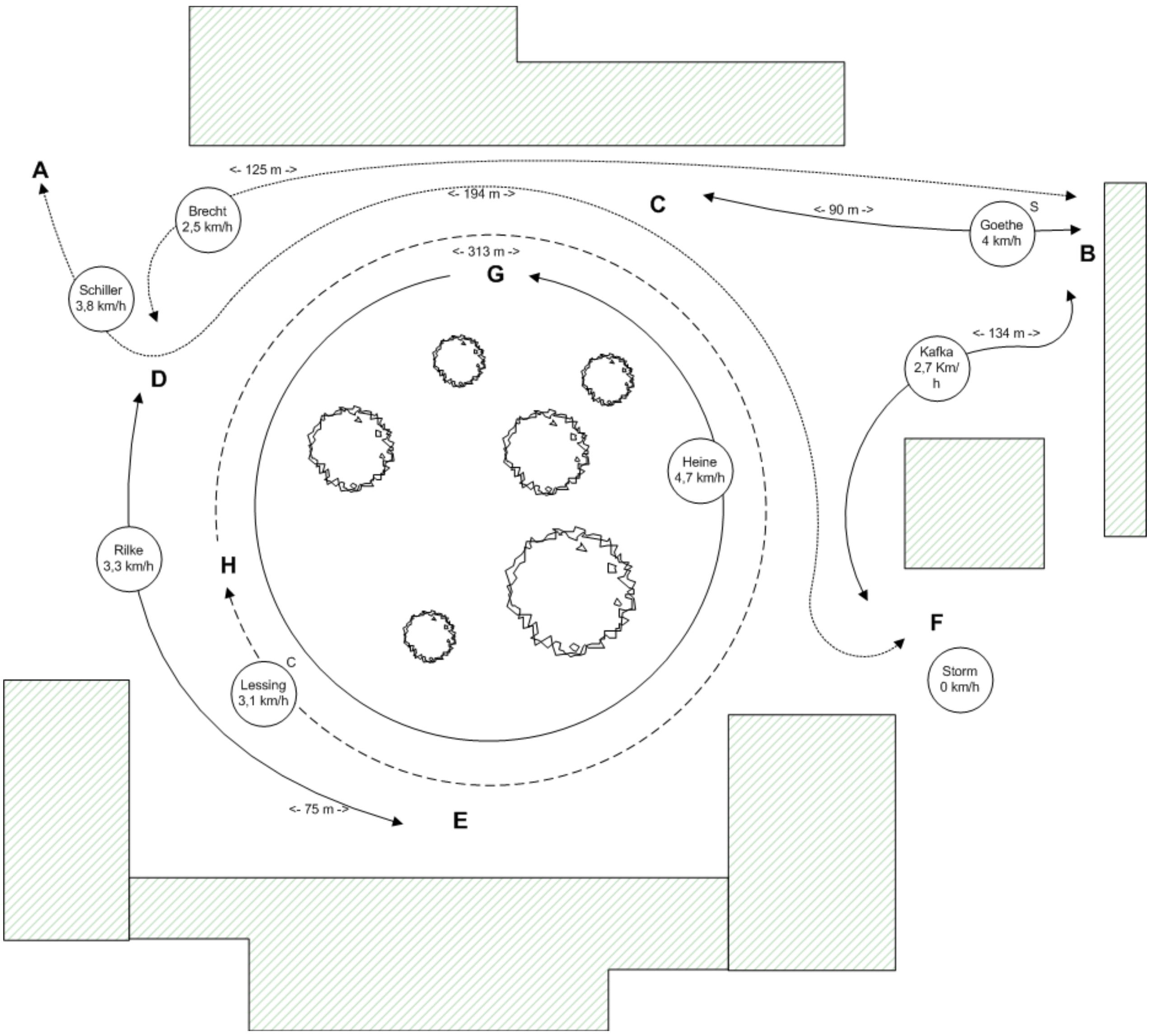


Figure 14.9.: reality test szenario scheme

14.6. UML

14.6.1. Description

We developed our testing-framework in terms of simplicity and portability. It should run in real-life and also in simulated real-life networks. The use of a simulated (virtual) network is obvious - test for a little network can be run on a single host. Several virtual

Figure 14.10.: reality tests

machine simulators exist - our choice was user-mode Linux (for more details see A.1.4). With our test systems (P4 - 2GHz) we can simulate up to five UMLs at one machine with reasonable speed.

14.6.2. Modifications

Our testing-framework was developed with regards to compatibility with quasi-reality and simulation - so that we can take the same scripts without any change. Anyhow a few changes have to be done: In our long-term test a loss of synchronization occurred so that topology changes did not take place at a defined time. As we use mackill to block the mac addresses in order to simulate a connection loss it was vital to let that change take place nearly at the same time. So we pick up an older idea of a modified UML-Switch. This allows us to block mac addresses at a centralized point.

The switch-daemon was modified because the original UML switch available does not support dynamic connections: all UML instances are linked with each other.

Now the switch daemon reads a configuration file when it is starting with an appropriate parameter (`uml_switch -wlan <filename>`). In this file the connections are listed which should exist. Here is an example:

```
default 0

merkur <-> venus 100
venus <-> erde 100
erde <-> jupiter 100
merkur <-> mars 100
mars <-> jupiter 100
venus <-> mars 100
```

Between `merkur` and `venus` 100% of the packets arrive, as well as between `venus` and `erde` and so on. The connections that are not given any explicit values use the default value. By this we realized a network which looks like the one shown in figure 14.11. The

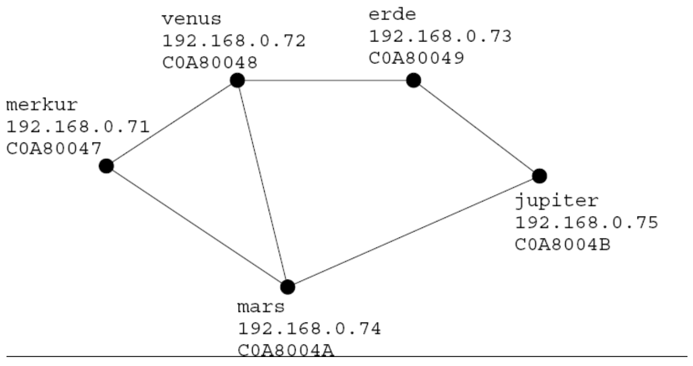


Figure 14.11.: example UML network

values between0 and 100 are possible, so for example `merkur <-> venus 60` will ensure that 60 % of the packets between `merkur` and `venus` arrive and 40 % are dropped. The dropped packets are randomly selected.

The switch internally works with mac addresses, therefore we provide a mapping from hostnames to mac addresses. For this purpose we declared an IP address for every hostname, which is mapped by the switch daemon to a mac address in a similar fashion as the UML machines do. If this is not sufficient, one can declare the mac address directly. An example is

```
merkur:  192.168.0.71
venus:   192.168.0.72
jupiter: a0:c1:34:91:2b:f2
saturn:  192.168.0.76
```

If the configuration file is changed, the switch will read it and the changes will immediately become valid.

However the packet dropping by the switch daemon does not sufficiently simulate wireless networks with different connection qualities and transmission rates. Nevertheless the networks can be easily simulated in which a computer is not connected with all other ones.

This is very good for testing the basic functionality of an algorithm. A realistic performance test is not possible with UML anyway. To adopt our test-scripts so that one UML node is the topology change server. This node initiates the topology change at a given script time. In order to do so this UML node has to access the host file system. The UMLs are capable of mounting the host file system (if this is properly compiled into the kernel). The command `mount none /mnt -t hostfs` mounts the whole filesystem under `/mnt` and now the node can copy predefined UML-switch config files to the UML-switch folder on the host fs. The UML-switch config immediately changes the topology.

14.6.3. Results

For testing in UML it is important to apply the SKAS patch to the kernel. With this patch the performance of the simulated machines increases significantly. The tests were performed with thrulay and 900 seconds for each run were given(except one testrun which was done using netperf to verify throughput results). Each test was performed five times and was averaged with the geometric mean. The 75 tests were conducted with a total duration of nearly 19 hours.

During the tests AODV remained unstable and crashed several times. OLSR had severe problems in finding new routes after a quick topology change.

14.6.4. TCP Results

Results of the TCP tests in user-mode linux are listed in tables 14.3,14.4,14.5,14.6 and are graphically represented in figure 14.13.

	AODV	BeeAdHoc	OLSR
topology change every 5 sec	2.472 Mbit/s	11.572 Mbit/s	1.167 Mbit/s
topology change every 10 sec	3.005 Mbit/s	13.467 Mbit/s	3.347 Mbit/s
topology change every 30 sec	3.347 Mbit/s	17.076 Mbit/s	9.870 Mbit/s
topology change every 60 sec	2.909 Mbit/s	18.553 Mbit/s	10.198 Mbit/s
static topology	2.314 Mbit/s	13.882 Mbit/s	6.016 Mbit/s

Table 14.3.: UML - Thrulay - Throughput(Mbit/s)

	AODV	BeeAdHoc	OLSR
topology change every 5 sec	2,29 Mbit/s	9,26 Mbit/s	1,30 Mbit/s
topology change every 10 sec	2,93 Mbit/s	13,87 Mbit/s	4,39 Mbit/s
topology change every 30 sec	3,28 Mbit/s	21,81 Mbit/s	11,3 Mbit/s
topology change every 60 sec	3,30 Mbit/s	19,24 Mbit/s	12,59 Mbit/s
static topology	2,20 Mbit/s	16,65 Mbit/s	7,55 Mbit/s

Table 14.4.: UML - Netperf - 900sec - Throughput(Mbit/s)

	AODV	BeeAdHoc	OLSR
topology change every 5sec	343,85 msec	248,55 msec	3350,22 msec
topology change every 10sec	244,58 msec	342,25 msec	4097,48 msec
topology change every 30sec	206,48 msec	257,80 msec	632,06 msec
topology change every 60sec	226,08 msec	262,32 msec	270,20 msec
static topology	276,04 msec	904,81 msec	2081,25 msec

Table 14.5.: UML - Thrulay - 900sec - average RTT

14.6.5. UDP Results

Results of the UDP tests in UML are listed in table 14.7.

	AODV	BeeAdHoc	OLSR
topology change every 5sec	2997	465	2258
topology change every 10sec	2892	279	2072
topology change every 30sec	2860	155	1809
topology change every 60sec	2322	121	1689
static topology	1802	34	1199

Table 14.6.: UML - Thrulay - 900sec - Control packets

	AODV	BeeAdHoc	OLSR
packets generated	14354	14024	14176
packets delivered	14336	14024	14174
percentage delivered	99,87%	100,00%	100,00%
avg packet delay	6,03 msec	1,49 msec	1,50 msec
avg session delay	298,92 msec	299,04 msec	299,46 msec
avg throughput	0,1970 Mbit/s	0,1921 Mbit/s	0,1942 Mbit/s
duration	298 sec	299 sec	299 sec
jitter	20,85 msec	21,32 msec	21,13 msec

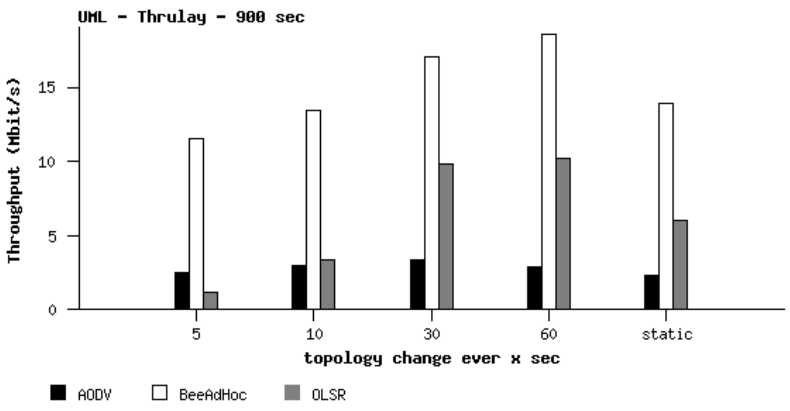
Table 14.7.: UML - sqtg - 300sec - static - 3 hops

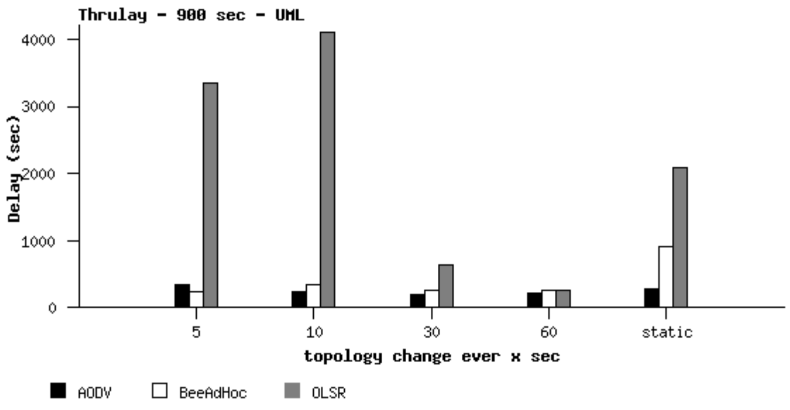
14.7. Quasireality

14.7.1. Results

Tests in Quasireality took more than four weeks (including setting nodes and developing scripts) - more than 130 hours of testing and analysis were needed to obtain these results. The tests in quasireality are divided into three main parts for TCP testing:

- TCP
  - 120sec
    - thrulay
      Five tests per network protocol - direct connection between sender and receiver, 1 hop between them and up to four hops.
  - 300 sec topology change every 5 seconds
    - D-ITG
  - 900 sec topology change every 30 seconds
    - D-ITG
    - netperf





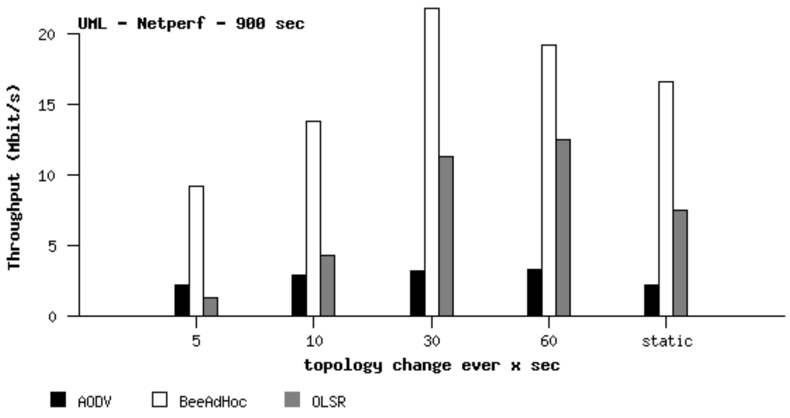


Figure 14.12.: graphical interpretation of UML tests - part 1

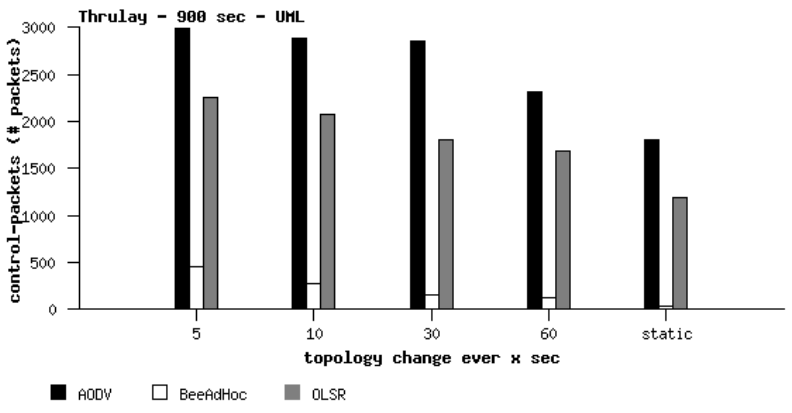


Figure 14.13.: graphical interpretation of UML tests - part 2

        * thrulay
        * batterytest
- UDP
    - 300sec
        * sqtg
        * D-ITG
        * D-ITG VoIP

14.7.2. TCP Results

Results of the TCP tests in quasireality are listed in tables 14.8 to 14.14 and are graphically represented in figures 14.14 to 14.19. Interesting is not only the average throughout, but also the stability of a connection. OLSR failed in obtaining a stable connection when topology changes are fast. For OLSR the average delay is correspondingly high. D-ITG didn't have these problems - the traffic generator lowers connection speed automatically.

Reasons for the differences between UML and quasireality test might be: a) BeeAdHoc parameters were optimized for UMLs (which are slightly different than real systems), b) the UML Switch was much faster than the wireless-LAN cards. The advantage of the UML tests is the nearly unlimited speed capacity of the switch. So the algorithms can show how there are able to scale to the available network traffic load. In the UML BeeAdHoc and OLSR archieved much higher throughputs than AODV, so AODV is unable to use the full bandwidth. In quasireality the available bandwidth is limited to the throughput of the wireless-LAN cards - all algorithms seem to make use of it (see figure 14.8).

	AODV	BeeAdHoc	OLSR	static
0 Hops	4,973 Mbit/s	4,814 Mbit/s	4,971 Mbit/s	4,838 Mbit/s
1 Hops	2,498 Mbit/s	2,471 Mbit/s	2,305 Mbit/s	
2 Hops	1,695 Mbit/s	1,653 Mbit/s	1,082 Mbit/s	
3 Hops	1,304 Mbit/s	1,267 Mbit/s	1,101 Mbit/s	
4 Hops	1,036 Mbit/s	1,013 Mbit/s	0,757 Mbit/s	

Table 14.8.: Quasireality - Thrulay - 120sec - Throughput

	AODV	BeeAdHoc	OLSR	static
0 Hops	366 msec	357 msec	366 msec	358 msec
1 Hops	692 msec	683 msec	720 msec	
2 Hops	975 msec	972 msec	1365 msec	
3 Hops	1170 msec	1174 msec	1439 msec	
4 Hops	1479 msec	1492 msec	2002 msec	

Table 14.9.: Quasireality - Thrulay - 120sec - avg RTT

	AODV	BeeAdHoc	OLSR
0 hops	236	1	134
1 hops	235	1	161
2 hops	239	1	228
3 hops	271	1	237
4 hops	275	2	217

Table 14.10.: Quasireality - Thrulay - 120sec - control-packets

	AODV	BeeAdHoc	OLSR
Netperf	0,190 Mbit/s	0,236 Mbit/s	0,129 Mbit/s
Thrulay	0,192 Mbit/s	0,251 Mbit/s	0,069Mbit/s
D-ITG	0,011 Mbit/s	0,016 Mbit/s	0,010Mbit/s

Table 14.11.: Quasireality - 900sec - Throughput

	AODV	BeeAdHoc	OLSR
Thrulay	5656,42 msec	5542,06 msec	18116,19 msec
D-ITG	15,27 msec	8,51 msec	12,72 msec

Table 14.12.: Quasireality - 900sec - avg delay

	AODV	BeeAdHoc	OLSR
Netperf	2616	1215	2787
Thrulay	2648	1884	2689
D-ITG	2681	1483	2800

Table 14.13.: Quasireality - 900sec - control-packets

	AODV	BeeAdHoc	OLSR
D-ITG	7883	15600	9316

Table 14.14.: Quasireality - 900sec -total data packets delivered

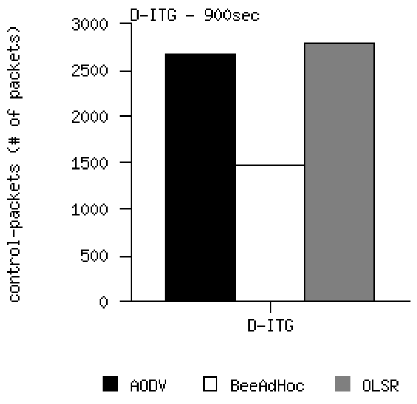


Figure 14.14.: Quasireality - 900sec - control-packets

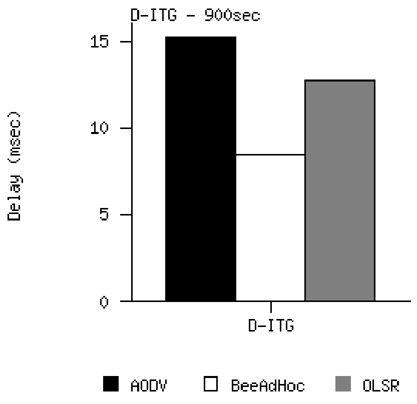


Figure 14.15.: Quasireality - 900sec - avg delay

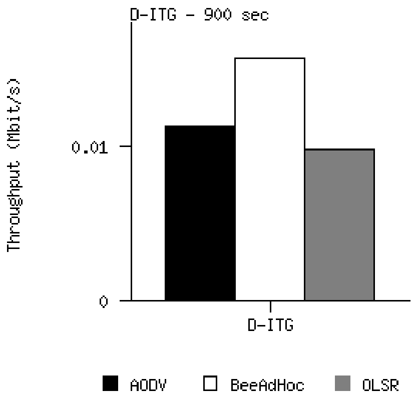


Figure 14.16.: Quasireality - 900sec - throughput

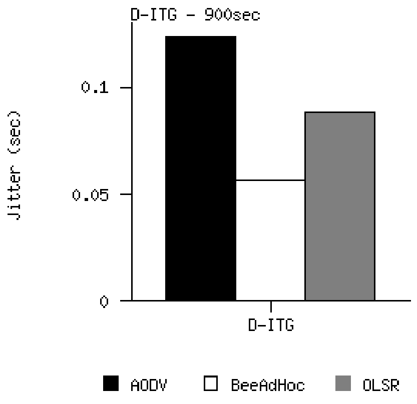


Figure 14.17.: Quasireality - 900sec - jitter

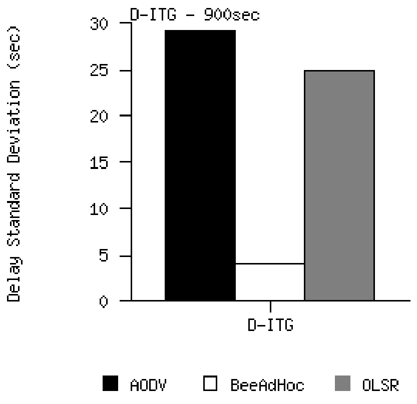


Figure 14.18.: Quasireality - 900sec - average standard deviation

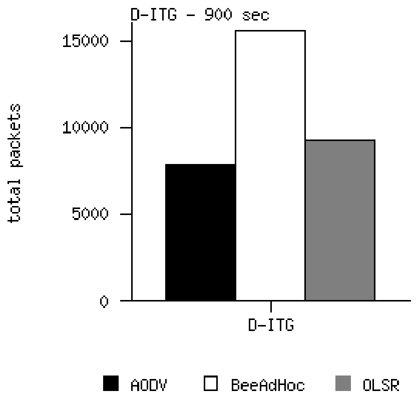


Figure 14.19.: Quasireality - 900sec - total data packets delivered

14.7.3. UDP Results

Results of the UDP tests in quasireality are listed in tables 14.15,14.16,14.17.

	AODV	BeeAdHoc	OLSR
percentage dropped	97,25%	67,98%	93,02%
avg. packet delay msec	4,755 msec	18,424 msec	12,379214 msec
avg. throughput	0,085 Mbit/s	0,052 Mbit/s	0,259 Mbit/s
delay standard deviation	2902,045 msec	15421,491 msec	6242,032 msec
duration	302 sec	260 sec	308 sec
jitter	22,408 msec	1241,814 msec	21,731 msec
control packets	651	855	538

Table 14.15.: Quasireality - D-ITG - 300sec - static - 3 hops

	AODV	BeeAdHoc	OLSR
packets generated	23927	23866	24059
packets delivered	23927	23866	24059
pervcentage delivered	100%	100%	100%
avg. session delay	1267,33 msec	692,33 msec	299,10 msec
avg. throughput	0,470 Mbit/s	0,578 Mbit/s	0,659 Mbit/s
duration	1267 sec	692 sec	299 sec
jitter	4,09 msec	6,08 msec	12,43 msec
control packets	824	142	572

Table 14.16.: Quasireality - sqtg - 300sec - static - 3 hops

	AODV	BeeAdHoc	OLSR
percentage dropped	0,00%	13.69%	6,87%
avg. packet delay	29,476 msec	4312,094 msec	25,335 msec
avg. throughput	0,074 Mbit/s	0,034 Mbit/s	0,068 Mbit/s
delay standard deviation	6,041 msec	12930,345 msec	10,277 msec
duration	300 sec	355 sec	300 sec
jitter	0,011 msec	280,725 msec	0,049 msec
control packets	684	653	611

Table 14.17.: Quasireality - D-ITG - 300sec - VoIP

14.7.4. Analysis

BeeAdHoc outperforms AODV and OLSR in nearly all parameter - indeed not as superior as under UML but still better.

14.7.5. Energy consumption

We measured the typical energy consumption of all three algorithms. To do so by logging the remaining battery capacity with

```
 cat
/proc/net/acpi/battery/BAT0/state
```

, which returns the mW/h remaining in the battery. The difference of the values between start and endtime is then the energy consumption(see 14.20). In figure 14.20 we compared the energy consumption of the three algorithms.

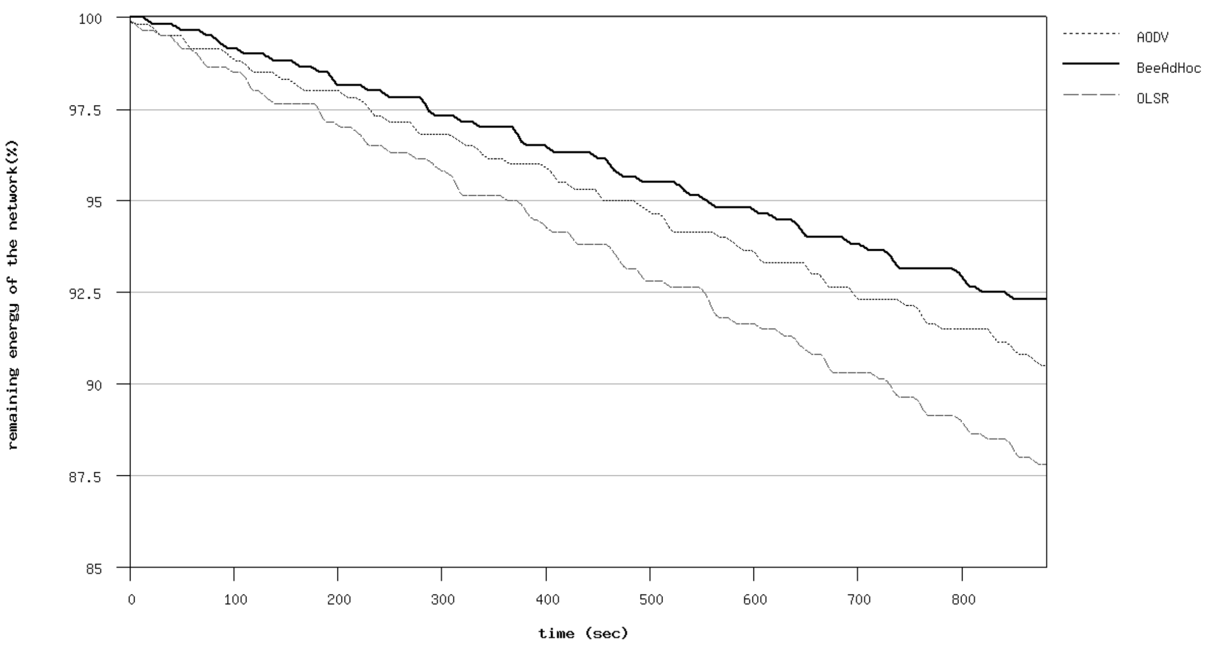


Figure 14.20.: Energy consumption

	AODV	BeeAdHoc	OLSR
Joule	14774,4 J	13413,6 J	14191,2 J

Table 14.18.: Joule consumption during 900sec testrun

14.8. Reality

14.8.1. mobility tests

Weather conditions: outdoor temperature 16 degree Celsius, 74% relative humidity, 1023 hPa barometric pressure, cloudy, no rain.

	AODV	Beeadhoc	OLSR
Throughput	0,730 MBit/s	0,627 MBit/s	0,882 MBit/S
avg. packet delay	8203 msec	2252 msec	2309 msec
controlpackets server in	1025	153	3818

Table 14.19.: Reality - Thrulay - 720sec - mobility

14.8.2. static tests

Weather conditions: outdoor temperature 25,1 degree Celsius, 53% relative humidity, 1015,4 hPa barometric pressure, sunny, no rain.

	AODV	Beeadhoc	OLSR
Throughput	0,416 Mbit/s	0,330 MBit/s	0,305 MBit/s
avg. packet delay	3986 msec	4117 msec	5078 msec
controlpackets server in	91	67	2537

Table 14.20.: Reality - Thrulay - 720sec - static

14.9. Analysis

As you can see the results speak for itself. But we want to outline some things here. First we have to say that TCP and UDP performance is very different. This is because the paket handling in UDP is different as explained before. Our UDP implementation (Swarms) is based on old work and no theoretical tests. The technical implementation is done but all parameters and conditions for sending such a paket is due to presumption.

And in addition all work for Swarms based on implementation work in UML environment. As for this you can see in table 14.7 that all algorithms performs nearly equal only AODV is a bit inferior. UDP results in the quasireality shows the weakness of our implementation and where more work has to be done (14.15, 14.16 and 14.17). In 14.16 you can see that our kernel modul performs well against 14.15 and 14.17.
More clearly are the TCP results which shows the strength of our implementation. When you compare the UML and reality results you can see that we lead with more distance in UML tests. This is because the network speed between the nodes in UML is not restricted and the algorithm can use all speed it can produce. In reality this speed is restricted throught hardware limitation of the WLAN cards. As you can see in table 14.8 also the static transfer (means no routing algorithm loaded) isn't faster. All results shows the superior design of BeeAdHoc (except UDP) within controll pakets, throughput and delay. In addition the power consumption of our algorithm is also lower as shown in figure 14.20.

A. Appendix

A.1. UML

When it comes to testing a system inside the kernel then the use of a normal testing kernel is quite time-consuming. A kernel or kernel module must be compiled and loaded into the memory andin case of changes in the kernel a reboot is necessary. Testing a kernel module is also often followed by a restart of the machine because a live unloading of the module may crash the kernel. For these situations a virtual machine is the solution because it works independent from the host system and in case of a crash it can be fastly restarted. Many solutions exist, but we choose user-mode-linux [Dik] because it is quite stable in its current state of development and is used in many other environments.

When starting a UML in a console it 'looks' just like a normal Linux booting up and running. One can see all startup messages and later a normal login prompt. A UML is a special compiled kernel which is compiled into one executable. The kernel needs to be patched for this (no need for kernel 2.6.10 and higher) and can be compiled with a `make linux ARCH=um` (don't forget to add the `ARCH=um` also to the make menuconfig). The resulting binary can now be executed under any kernel such as a normal user-space program. Kernel modules for the UML must also be also compiled with the special `ARCH=um` switch. The hostsystem running the UML should be patched with SKAS. This patch separates the memory usages for the different virtual systems which results in relatively faster simulation speeds. For our testing we used kernel 2.6.9 for the UML and kernel 2.6.7 with SKAS patch for the host system.

A.1.1. Filesystem

UMLs consist of two main components: the kernel executable and the imagefile which contains the whole filesystem. The path to the imagefile is given as a parameter to the command-line and the UML will then use this file as a primary hdd. Every UML needs its own filesystem, therefore it requires a large memory and the sync process between these images is time-consuming. The solution is copy-on-write (COW), which is one master filesystem used by every UML for booting. Any changes that are individually made during the runtime are written in a special cow file (one for every virtual machine). These COW files must not be created or modified manually because the UML takes care of this. An example of booting a machine with COW support is shown later.

A simple filesystem is created with some special tools or one can get predefined images (e.g. from [Dik]) and adjust them (like we did).

To transfer files between the UMLs and the host two possibilities exist: copying files over the network or mounting the host filesystem into a UML by using:

```
mount none /mnt -t hostfs
```

A.1.2. Network

There are several possibilities to connect different UMLs with each other. The best choice for us is the usage of the UML-switch daemon. This is a userspace program which runs on the host besides the virtual machines. When starting the UMLs the switch `eth0=daemon` makes sure that the virtual eth0 device in the UML is connected to the UML-switch which now acts as a standard network switch. If now a UML wants to send data to another machine the switch conncts the correct machines. To connect this virtual network with a real one the TAP-Interface on the host can be used. This TAP interface is used by userspace programs to receive and send packets. On the host system a TAP interface and an ethernet interface are bridged so it is possible to connect several real host, which are running several virtual host, in order to simulate a bigger network. All required commands can be found at URL: `http://user-mode-linux.sourceforge.net/networking.html`.

A.1.3. Starting User-mode Linux

In order to add files to all machines the base machine must be started like this:

```
linux ubd0=images/Debian-3.0r0\_aodv.ext2 eth0=daemon devfs=mount
con1=none
```

In this case the compiled UML executable is named `linux` and the image `Debian-3.0r0_aodv.ext2` is located in the directory `images`. The next command is used to boot a UML from the base image and to write changes in a cow file.

```
linux mem=16M
ubd0=images/cow-merkur_aodv,images/Debian-3.0r0_aodv.ext2
umid=merkur eth0=daemon,FE:FD:C0:A8:00:00 devfs=mount con1=none &
```

Most switches are self-explanatory and you can specify how much RAM should be assigned to a UML (`mem=`), give the UML an ID (`umid=`) and let the eth0 interface connect to the switch-daemon with a specific mac (`eth0=`).

A.1.4. Dealing with problems

Working with user-mode linux is quite simple but at times brings specific problems as well. One main rule is to make backups of the filesystem for the base UML. It can happen that the filesystem is not fixable because of kernel crashes.

Virtual machines should always be halted properly and should be never booted from the same base filesystem (except when using COWs).

If machines abruptly stop working a look at the error messages might reveal that very often the filesystem is blocked by another process. This can happen if a UML has crashed but did not get removed from the memory. On can kill dead UMLs with `pkill linux` and then start again.

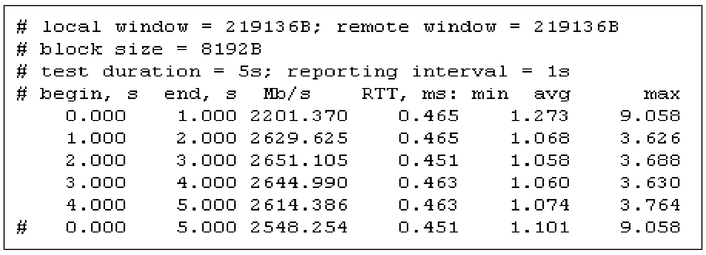
Setting up the UML can be quite tricky. We played with the `devfs=mount` and `con1=none` switches because they cause trouble on some systems. When compiling modules for the UML one needs to point to the right kernel directory and the UML executable must be previously compiled from this source. When getting problems during compiling one need to ensure that the commands `make clean` or `make mrproper` are helpful. Also check if the symbolic link under `include/` is pointing to `include/asm-um`.

For proper testing under X the usage of xterm has the advantage that every virtual machine can be started automatically in its own window. Additionally the window title can be freely assigned so that every window has the machines name on on. To start any program in a new xterm session with a specific title the following command is used:

```
xterm -T <title> -e <program>
```

A.2. Thrulay

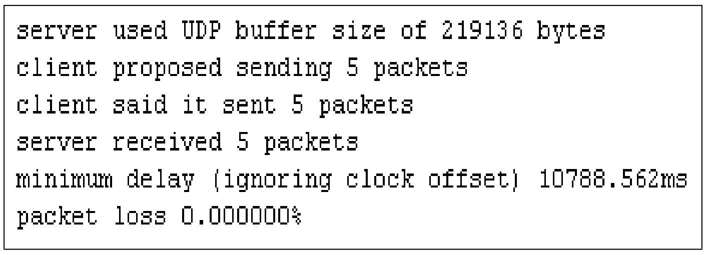
'thrulay' is a 'client-server-solution' like many other traffic generators. On server site there is the program 'thrulayd' which listen on a specific port. The server sends back a data packet with a timestamp to the client which calculates the values 'throughput', 'packet delay' and 'packet delivery ratio'. The client sends data packets in a certain inteval of a certain size. The client is the program 'thrulay'. The client also shows the user the results of the test. Also one can adjust in which interval the output is shown. In figure A.2 one can see a typical output of 'thrulay'.



```
# local window = 219136B; remote window = 219136B
# block size = 8192B
# test duration = 5s; reporting interval = 1s
# begin, s  end, s  Mb/s     RTT, ms: min  avg      max
     0.000     1.000 2201.370    0.465    1.273    9.058
     1.000     2.000 2629.625    0.465    1.068    3.626
     2.000     3.000 2651.105    0.451    1.058    3.688
     3.000     4.000 2644.990    0.463    1.060    3.630
     4.000     5.000 2614.386    0.463    1.074    3.764
#    0.000     5.000 2548.254    0.451    1.101    9.058
```

Figure A.1.: 'thrulay' tcp test

In TCP - Protokoll one can measure the parameters 'throughput' and 'round trip time'. In UDP protocol 'thrulay' can measure the parameteres 'packet delay' and 'packet de-



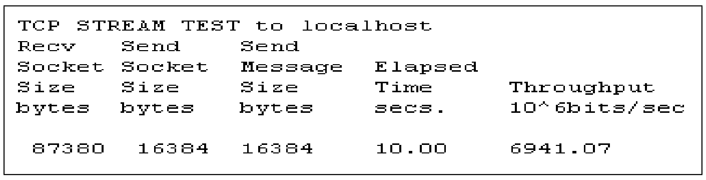
```
server used UDP buffer size of 219136 bytes
client proposed sending 5 packets
client said it sent 5 packets
server received 5 packets
minimum delay (ignoring clock offset) 10788.562ms
packet loss 0.000000%
```

Figure A.2.: 'thrulay' udp test

livery'.

A.3. Netperf

The scond trafficgenerator we have used is 'netperf'. Like 'thrulay' 'netperf' can also generate transport layer protocols like TCP and UDP. The trafficgenerator can measure 'average throughput' and 'packet delivery ratio'. Within 'netperf' serveral scripts are



```
TCP STREAM TEST to localhost
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384   16384    10.00    6941.07
```

Figure A.3.: 'netperf' tcp test

available which have preset settings with TCP and UDP.

A.4. FTP

A.5. Wget

Wget is a well known tool under linux to retrieve files from the Web. It is capable of transferring HTTP, HTTPS and also FTP data in an non-interactive process. As for our test framework, it was used to retrieve standardized testfiles from machines in the testing network.

```
wget -t 0 -T 10 -w 0 -O /dev/null -a ./Log/wgetlog
ftp://ftpdummy:test\@192.168.0.71/testfile
```

The paramete rset specifies number of retries (-t) and the timeout (-T) which is important to guarantee uninterrupted testing in scenarios where hosts are not available during topology switching.

A.6. Ping

A simple classic tool that allows an easy way of testing connections between nodes is ping. During our tests it was used to detect errors in the algorithms, i.e. a significantly greater packet loss on an undisturbed line showed us that packets are corrupted. After this the usage of tcpdump A.7 we could spotted the error. A typical output looks like this:

```
PING erde (192.168.0.73): 56 data bytes 64 bytes from 192.168.0.73:
icmp_seq=0 ttl=63 time=541.0 ms 64 bytes from 192.168.0.73:
icmp_seq=1 ttl=63 time=5.7 ms 64 bytes from 192.168.0.73: icmp_seq=2
ttl=63 time=5.9 ms 64 bytes from 192.168.0.73: icmp_seq=3 ttl=63
time=8.1 ms 64 bytes from 192.168.0.73: icmp_seq=4 ttl=63 time=8.1
ms 64 bytes from 192.168.0.73: icmp_seq=5 ttl=63 time=7.3 ms 64
bytes from 192.168.0.73: icmp_seq=6 ttl=63 time=11.2 ms

--- erde ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 5.7/83.9/541.0 ms merkur:~#
```

Here it is important to notice that the first ping takes 541ms whereas the following ones are up to 100 times faster. This is explained with the fact that at the start up the route to erde was not known to merkur - so a route has to be found first.

For more route information one can take a look at traceroute (A.7).

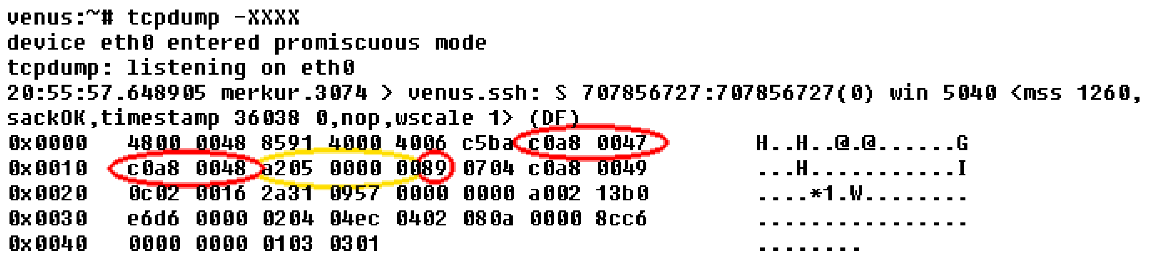
A.7. Traceroute

A small and simple tool that allows quick testing if the algorithms routing is correct. Simply start it by `traceroute <host>`. A typical screen dump looks like this:

```
traceroute to erde (192.168.0.73), 30 hops max, 38 byte packets
 1  venus (192.168.0.72)  1.452 ms  533.281 ms  1.210 ms
 2  erde (192.168.0.73)  7.385 ms  7.520 ms  6.773 ms
```

A.8. TCPDump

It is indispensable for bug indentification. It allows to inspect any network packet that flows through a specified network card. So one can check if for modified packets the checksum is still correctly calculated or if the ip options are set correctly.



```
venus:~# tcpdump -XXXX
device eth0 entered promiscuous mode
tcpdump: listening on eth0
20:55:57.648905 merkur.3074 > venus.ssh: S 707856727:707856727(0) win 5040 <mss 1260,
sackOK,timestamp 36038 0,nop,wscale 1> (DF)
0x0000   4800 0048 8591 4000 4006 c5ba c0a8 0047        H..H..@.@......G
0x0010   c0a8 0048 a205 0000 0089 0704 c0a8 0049        ...H...........I
0x0020   0c02 0016 2a31 0957 0000 0000 a002 13b0        ....*1.W........
0x0030   e6d6 0000 0204 04ec 0402 080a 0000 8cc6        ................
0x0040   0000 0000 0103 0301                            ........
```

Figure A.4.: tcpdump example output

Figure A.4 shows a typical example how tcpdump can be used to extract important information from ip packets. The info line shows us that the packet started from merkur (192.168.0.71) and was destinated for venus (192.168.0.72). Inside the ip package these addresses are coded as the hexadecimal values c0a80047 for the source ip 192.168.0.71 and c0a80048 for the destination ip 192.168.0.71. After the destination address the ip options field is filled with some important options. First there is the BeeAdHoc option - the option number is 162 which is encoded as *a2*. The length of the BeeAdHoc field is 5 and there are currently no values stored (all are zero). As BeeAdHoc currently uses sourcerouting the next entry in the options field is 89 which 137 in decimal and is the ip option number for strict source routing.

A.9. Scenario- Editor

Scenario editor allows us to generate testing scenarios with static and dynamic topologies. We don't distinguish between two nodes which are close to each other and two nodes far away from each other but with connection. It means the quality of one connection is the same as far as the connections exists. That's why we can divide the simulation in different phases with the same set of connections and then repeating them.

A.9.1. Standard Run

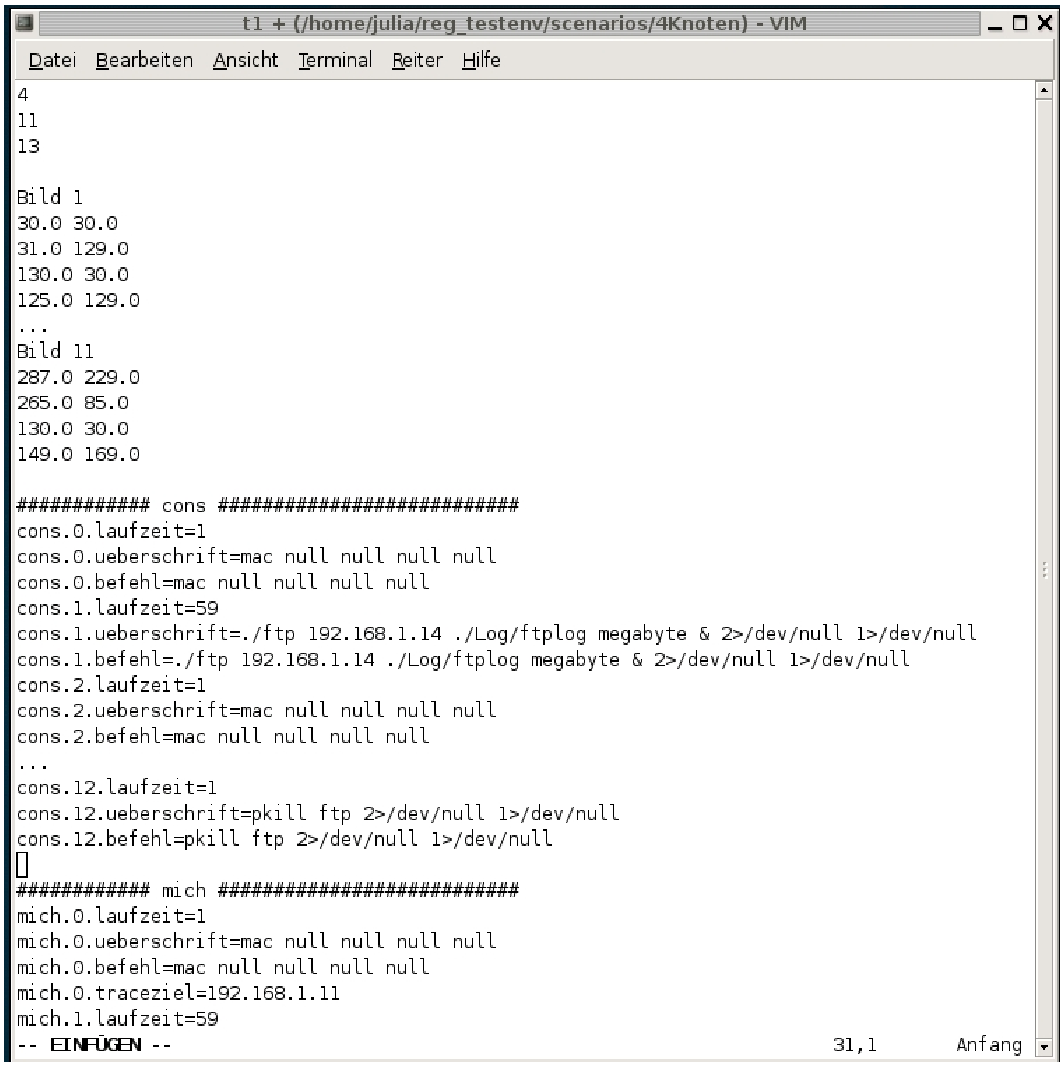
One can prepare a scenario in the following:

1. start the scenario-editor
2. press `insert` to insert a new phase. You see a pop-up-window. You have to fill the duration field (insert the time in seconds). If you will start some transfers you have to enter the node number and the instruction for this node. Figure A.6 shows an example. If you want to start another transfer simultaneously press `gleichzeitig` and fill the appropriate fields. When you are ready press `ok`. Now you can change the positions of the nodes. The position of a node is encoded as an instruction which contains the information about the connection state (connected or disconnected) of this node to other nodes. It means if we start a transfer we have two instructions for one node in one phase. That's why we divide such phases in two steps. The first step executes the transfer instruction and the second one the connection instruction.
3. It can be necessary to execute some instructions at the end of the test (e.g. pkill ftp) You can do it while you repeate the 2. point and insert the desired instructions manualy. If all instructions are the same you can bind the `End`-button with this instruction. At the moment it is `pkill ftp`, because we tested ftp-transfers first.

A.9.2. Methods

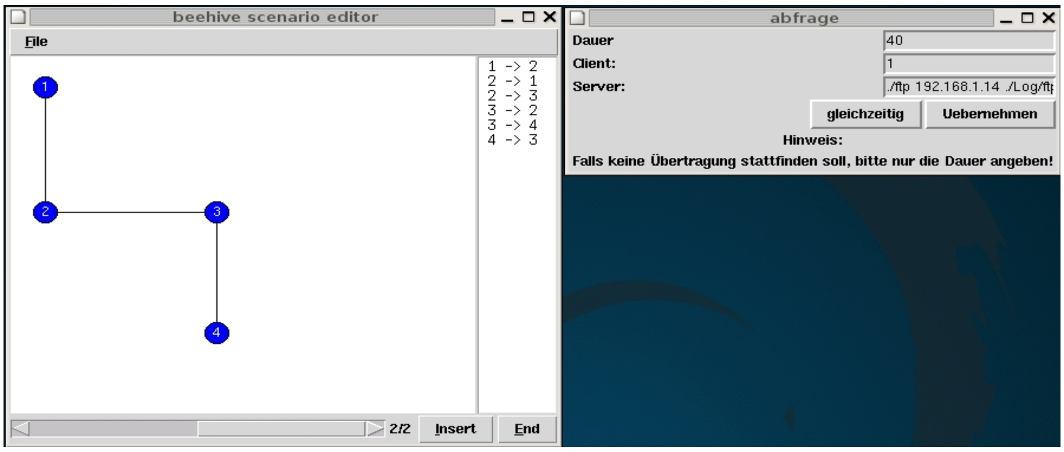
Following important methods are implemented:

`newtime`
It proofs whether the input is a number.
`update_lines`


t1 + (/home/julia/reg_testenv/scenarios/4Knoten) - VIM
Datei Bearbeiten Ansicht Terminal Reiter Hilfe

```
4
11
13

Bild 1
30.0 30.0
31.0 129.0
130.0 30.0
125.0 129.0
...
Bild 11
287.0 229.0
265.0 85.0
130.0 30.0
149.0 169.0

############ cons ###########################
cons.0.laufzeit=1
cons.0.ueberschrift=mac null null null null
cons.0.befehl=mac null null null null
cons.1.laufzeit=59
cons.1.ueberschrift=./ftp 192.168.1.14 ./Log/ftplog megabyte & 2>/dev/null 1>/dev/null
cons.1.befehl=./ftp 192.168.1.14 ./Log/ftplog megabyte & 2>/dev/null 1>/dev/null
cons.2.laufzeit=1
cons.2.ueberschrift=mac null null null null
cons.2.befehl=mac null null null null
...
cons.12.laufzeit=1
cons.12.ueberschrift=pkill ftp 2>/dev/null 1>/dev/null
cons.12.befehl=pkill ftp 2>/dev/null 1>/dev/null

############ mich ###########################
mich.0.laufzeit=1
mich.0.ueberschrift=mac null null null null
mich.0.befehl=mac null null null null
mich.0.traceziel=192.168.1.11
mich.1.laufzeit=59
-- EINFÜGEN --                                                     31,1          Anfang
```

Figure A.5.: The output file of the



```
1 -> 2
2 -> 1
2 -> 3
3 -> 2
3 -> 4
4 -> 3
```

Dauer: 40
Client: 1
Server: ./ftp 192.168.1.14 ./Log/ft

Hinweis:
Falls keine Übertragung stattfinden soll, bitte nur die Dauer angeben!

Figure A.6.: Scenario editor and abfrage

Every time a node is moved we have to update the connections of this node. For doing so we have to take the new positions and the new connections into account.
`insert`
This method executes `abfrage` first so that it will have all the nessessary values. Then it copies the node positions from the last phase and executes `update_lines`.
`save`
This procedure saves the prepared scenario for tests in quasireality. You can open this file later.
`export`
The prepared scenario will be saved for UML. It means we make a `wlan_conf` file for each phase and save the number of phases and the durations in the scenario file.
`uebernehmen`
This procedure takes care about the phases and steps. The inputs of the `abfrage` window will be saved correct.

A.9.3. Data structures

The important data structures are:

`clist`
It is a canvas list. Every entry describes one phase of the scenario.
`cnow`
It is the canvas for the phase that is actually shown.
`zeitList`

It is a time list with the durations of steps as enrties.
`tlist`
It is a time list too, but with the duration of phases as entries.

A.10. ITG

The internet traffic generator (ITG) was founded [SG] from as a UDP and TCP traffic generator which is available for the platforms Windows and Linux. On the sender side you must start the program "ITGSend" and on the receiver side you must start the program "ITGRecv". For evaluating you must use the program "ITGDec". With this traffic generator you can start several flows on a sender to multiple destination. Also you can use the protocols "Telnet" and "VoIP".
ITG can measure several parameters which are in a certain file after evaluating with "ITGDec":

```
----------------------------------------------------------
Flow number: 1
From 192.168.2.24:32803
To    192.168.2.25:8999
----------------------------------------------------------
Total time               =    907.946553 s
Total packets            =         44734
Minimum delay            =      0.088834 s
Maximum delay            =     33.431760 s
Average delay            =      8.626932 s
Average jitter           =      0.038758 s
Delay standard deviation =      4.792223 s
Bytes received           =      22903808
Average bitrate          =    201.807544 Kbit/s
Average packet rate      =     49.269420 pkt/s
Packets dropped          =             0 (0.00 \%)
----------------------------------------------------------


----------------------------------------------------------
****************  TOTAL RESULTS   ******************
----------------------------------------------------------
```

```
Number of flows          =          1
Total time               =   907.946553 s
Total packets            =        44734
Minimum delay            =     0.088834 s
Maximum delay            =    33.431760 s
Average delay            =     8.626932 s
Average jitter           =     0.038758 s
Delay standard deviation =     4.792223 s
Bytes received           =     22903808
Average bitrate          =   201.807544 Kbit/s
Average packet rate      =    49.269420 pkt/s
Packets dropped          =            0 (0.00 \%)
Error lines              =            0
----------------------------------------------------------
```

A.11. STG

The scientific traffic generator (STG) founded by [WF05] is a UDP based traffic generated which can work in a session-oriented mode and in a session-less mode. In the session-less mode you must specify the parameters start time, duration, MPIA (mean packet inter arrival time), the packet size and the destination:

```
tg -t <start time> -d <destination> -e <duration> -p <MPIA> -s <packet size>
```

In the session-oriented mode you must also specify the parameters session size and MSIA (mean session inter arrival time).

```
tg -t <start time> -d <destination> -e <duration> -p <MPIA> -s <packet size>
-m <MSIA> -S <Session size>
```

For STG you have to start two applications. On the receiver side you must the program tgsink which collects the data from the program on the sender side. The program there is the traffic generator tg self. If the test are over you can evaluate the file from tgsink with the program tgevaluated. After using tgevaluated the file looks like this:

```
packets generated: 120487,
packets delivered: 8912,
percent delivered: 7.396648e+00,
average packet delay: 5.239057e+05,
average session delay: 2.780659e+01,
jitter: 2.458988e+02,
90th packet delay: 0.000000e+00,
90th session delay: 4.837001e+01,
90 jitter: 0.000000e+00
-----------------------------------------------------------------

packets generated: 120487,
packets delivered: 8912,
percent delivered: 7.396649e+00
average packet delay: 5.239057e+05 ms,
average session delay: 2.780659e+01 sec
jitter: 2.458988e+02 ms
90-packet delay: 0.000000e+00 ms,
90-jitter: 0.000000e+00,
90-session delay: 4.837001e+01
packets that entered a loop: 0,
percent of looping packets: 0.000000e+00,
average hop count: 3.978344e+00
average throughput: 5.659465e-01 Mbit/sec,
duration: 129 sec
sessions completed: 0,
sessions incomplete: 79,
session completion ratio: 0.000000e+00
```

You can also see the parameters STG can measure in this file.

A.12. Mackill

Mackill is a little tool for dropping IP Packets which where sent by an host having a certain hardware address. For working a little kernel patch was made by us to use mackill-function in the kernel. The tool was written for the 2.4.x Linux kernels but because the "dev.c" did not changed form 2.4.x to 2.6.x the patch can easily done. Tool was created by the University of Uppsala [Nora]. For patching the kernel the function "process_backlog" must be changed:

```
extern int (*mac_kill)(struct sk_buff *)=NULL;
EXPORT_SYMBOL_NOVERS(mac_kill);
static int process_backlog(struct net_device *backlog_dev, int *
    budget)
{
        .
        for (;;) {
                .
                dev = skb->dev;

                if (mac_kill && (*mac_kill)(skb)) {
                        kfree_skb(skb);
                        dev_put(dev);
                        continue;
                }

                netif_receive_skb(skb);

                dev_put(dev);

                work++;
                .
        }
        .
}
```

Listing A.1: Changing process_backlog

If the patch is applied and making and installing the new kernel, you can compile the mackill module against the source and the then use it.
For killing all packets from a host, you use the command:

```
> echo -XX:XX:XX:XX:XX:XX. > /proc/net/mackill
```

For delete a host form this ban list, you have to use the command:

```
> echo +XX:XX:XX:XX:XX:XX. > /proc/net/mackill
```

With the help of mackill you can create a quasireality like described in [Nora].

A.13. WSCCS

WSCCS is an abbreviation for Weighted Synchronous Calculus of Communicating Systems and strictly sticks to the concept of *locality and decentralised organisation* of agent systems. To describe a system's environment in WSCCS is rather clumsy because WSCCS has not been developed for this purpose. For analysis purposes one can use the *Probability Workbench*, which is designed for simulation of WSCCS agents according to their (WSCCS) semantics.
This appendix is just an introduction to WSCCS, which therefore leaves out elements of WSCCS which are not important for understandings of the BeeAdHoc or bee agents (see chapter 2 for the bee or chapter 4 for the BeeAdHoc agents).[1]

A.13.1. Syntax

To describe a WSCCS agent there are *actions*, *weights* and *labels*. A *label* represents one agent and starts with a capital letter, for example: $Packer(n, d, wt)$.
actions describe interaction and communication between the agents. Each action always has a correspondent responding action. Formally said the set of actions is an abelian group, which has the identity $\surd$ and the inverse, which is the response to an action a, is denoted by $\overline{a}$. If several actions, say a and b, are performed parallelly, but are represented sequentially as $a * b$; but they do not consequently are performed in the noted order ($*$ is a communicative relation). This should be remembered when dealing with parallel agents.
Weights are used to express priorities and probabilities and are noted in the following way: $W = \{ \omega_i \mid \omega_i = n\omega^k \; n\epsilon Z^+ \;, \; k \geq 0 \}$ where n is a relative frequency[2] and ω^k a priority. A bigger k means a greater priority. There are the following rules for priorities:

$$\begin{aligned} n\omega^k + m\omega^l &= n\omega^k \text{ for } k > l \\ n\omega^k + m\omega^k &= (n+m)\omega^k \\ n\omega^k * m\omega^l &= (nm)\omega^{k+l} \end{aligned} \tag{A.1}$$

[1] A more detailed and formal description of WSCCS is in the appendix of [Sum00].
[2] Since an agent can be confined in its actions by $| \{ A \}$, the probabilities might cause problems, because they may not sum up to 1 after the confinement.

An agent E is defined by induction:

$$E ::= 0 \mid A \mid a.E \mid \sum_{i\epsilon I} \omega_i : E_i \mid E_1 \times E_2 \mid (E \mid L) \mid \Theta(E) \tag{A.2}$$

The meaning of the symbols is:

0	an agent, which cannot perform any action
A	an agent labelled by A
$a.E$	an agent, which can perform an action a and then becomes an agent E
$\sum_{i\epsilon I} : E_i$	an agents which acts like E_K with a probability of $\frac{\omega_k}{\sum_{i\epsilon I} \omega_i}$[3]
$E_1 \times E_2$	a synchronized, parallel composition of the agents E_1 and E_2
$E \mid L$	an agent E , which is just allowed to perform actions of the set L
$\Theta(E)$	an agent which can only perform all the actions of E with the *highest priority*

Notes

* $A \equiv 1 : \surd.\surd.B$ is an abbreviation of $A \equiv 1 : \surd.(1 : \surd.B)$
* a parallel composition $a * b$ of actions can be also written as ab
* In praxis one may use the agent $Deleted \equiv 1 : \surd.Deleted$. In the formal definition of WSCCS this has to expressed by $F = \{x_k = 1 : \surd.x_k\}$ and $Deleted \equiv fix_k(F)$. The two expressions have the same meanings, but the latter one is needed to define a semantic in a formal way. The fix point rule is not understandable in first glance and is not needed for understanding the agents. Therefore it has been skipped.

A.13.2. Semantics

The semantics consists of rules for deduction steps, which describe the transitions an agent can perform. A rule looks like:

$$\frac{\text{hypothesis}}{\text{conclusion}}$$

[3] The agent $\sum_{i\epsilon I} : E_i$ is not forced to choose only the highest priority unless we write $\Theta(\sum_{i\epsilon I} : E_i)$. In the first case the priorities are ignored and just the relative frequencies do count. The letter agent only performs the actions with highest priority according to their relative frequencies. For example the agent $\omega : eat.A + \omega : work.A + 1 : sleep.B$ performs either actions with a probability of $\frac{1}{3}$, but the agent $\Theta(\omega : eat.A + \omega : work.A + 1 : sleep.B)$ just works and eats both with a probability of $\frac{1}{2}$.

An empty hypothesis means that the conclusion always holds; applying the rules to an agent one can deduce his behaviour. This is made clearer in the examples below.

$$\frac{}{a.E \xrightarrow{a} E}\ \text{(Act)} \qquad \frac{}{\sum\{\ \omega_i : E_i \mid i\epsilon I\ \} \overset{\omega_i}{\rightsquigarrow} E_i}\ \text{(Wght)}$$

$$\frac{E \xrightarrow{a} \acute{E} \wedge F \xrightarrow{b} \acute{F}}{E \times F \xrightarrow{ab} \acute{E} \times \acute{F}}\ \text{(Par1)} \qquad \frac{E \overset{\omega}{\rightsquigarrow} \acute{E} \wedge F \overset{\nu}{\rightsquigarrow} \acute{F}}{E \times F \overset{\omega\nu}{\rightsquigarrow} \acute{E} \times \acute{F}}\ \text{(Par2)}$$

$$\frac{E \xrightarrow{a} \acute{E} \wedge F \overset{\omega}{\rightsquigarrow} \acute{F}}{E \times F \overset{\omega}{\rightsquigarrow} E \times \acute{F}}\ \text{(Par3)} \qquad \frac{E \overset{\omega}{\rightsquigarrow} \acute{E} \wedge F \xrightarrow{b} \acute{F}}{E \times F \overset{\omega}{\rightsquigarrow} \acute{E} \times F}\ \text{(Par3)}$$

$$\frac{E \xrightarrow{a} \acute{E}\ a\epsilon A}{does_A(E)}\ \text{(Does1)} \qquad \frac{E \overset{\omega}{\rightsquigarrow} \acute{E} \wedge does_A(\acute{E})}{does_A(E)}\ \text{(Does2)}$$

$$\frac{E \xrightarrow{a} \acute{E}\ a\epsilon A}{E \mid A \xrightarrow{a} \acute{E} \mid A}\ \text{(Res1)} \qquad \frac{E \overset{\omega}{\rightsquigarrow} \acute{E} \wedge does_A\acute{E}}{E \mid A \overset{\omega}{\rightsquigarrow} \acute{E} \mid A}\ \text{(Res2)}$$

$$\frac{E \xrightarrow{a} \acute{E}}{\Theta(E) \xrightarrow{a} \Theta(\acute{E})}\ \text{(Pri1)}$$

$$\frac{E \overset{n\omega^k}{\rightsquigarrow} \acute{E}\ \text{(where } k \geq \acute{k} \text{ for all } E \overset{m\omega^{\acute{k}}}{\rightsquigarrow} \hat{E}\ )}{\Theta(E) \overset{n}{\rightsquigarrow} \Theta(\acute{E})}\ \text{(Pri2)}$$

(A.3)

Communication In WSCCS communication between two agents A and B is described in the way that A can perform an action a and B can respond to A with $\overline{a}$. B can be forced to answer A by allowing the agent $A \times B$ to perform only a $\surd$ action.

Example A packer searches for a forager:
$(Packer(n, d, wt) \times ForagerAtHive(n, d, i_{n,d}, RV, T_{FW})) \mid \{\surd\}$
The two act parallelly ($\times$)and should communicate $\mid \{\surd\}$. In the next step the forager should recruit the packer. This can be formulated in WSCCS by:

$$\begin{aligned}&(Packer(n, d, wt) \times ForagerAtHive(n, d, i_{n,d}, RV, T_{FW})) \mid \{\ \surd\ \} \xrightarrow{\surd} \\ &(Recruit(n, d) \times RecruitingForager(n, d, i_{n,d}, RV, T_{FW} + 1, dt)) \mid \{\ \surd\ \}\end{aligned} \tag{A.4}$$

The agents *Packer* and *ForagerAtHive* have different possibilities to behave; (see equation 4.9 on page 28 and equation 4.1 on page 23)but they can only communicate in one way:

$$\frac{}{\begin{array}{l} \mathit{Packer}(n,d,wt)\overset{\omega^2}{\rightsquigarrow} \\ \mathit{searchForager}_{(n,d)}.\mathit{Recruit}(n,d) \end{array}}\,(\text{Wght}) \tag{A.5}$$

and

$$\frac{}{\begin{array}{l} \mathit{ForagerAtHive}(n,d,i_{n,d},RV,T_{FW}))\overset{\omega^2}{\rightsquigarrow} \\ \overline{\mathit{searchForager}_{(n,d)}}.\mathit{RecruitingForager}(n,d,i_{n,d},RV,T_{FW}+1,dt) \end{array}}\,(\text{Wght}) \tag{A.6}$$

This is one possible application of the (Wght) rule on the two agent, where an action of each agent is chosen arbitrarily. However the weights ω^2 do not play any role yet.

The complete deduction or *proof tree*[4], which describes the communication between a *Packer* and a *ForagerAtHive*:

$$\frac{\frac{\begin{array}{l} \mathit{searchForager}_{(n,d)}\overline{\mathit{searchForager}_{(n,d)}}=\surd \\[2ex] \bigwedge \text{(A.5)} \bigwedge \dfrac{}{\begin{array}{l} \mathit{searchForager}_{(n,d)}.\mathit{Recruit}(n,d) \\ \xrightarrow{\mathit{searchForager}_{(n,d)}} \\ \mathit{Recruit}(n,d) \end{array}}\,(\text{Act}) \\[2ex] \bigwedge \text{(A.6)} \bigwedge \dfrac{}{\begin{array}{l} \overline{\mathit{searchForager}_{(n,d)}}.\mathit{RecruitingForager}(n,d,i_{n,d},RV,T_{FW}+1,dt) \\ \xrightarrow{\overline{\mathit{searchForager}_{(n,d)}}} \\ \mathit{RecruitingForager}(n,d,i_{n,d},RV,T_{FW}+1,dt) \end{array}}\,(\text{Act}) \end{array}}{\begin{array}{l} (\mathit{Packer}(n,d,wt)\times \mathit{ForagerAtHive}(n,d,i_{n,d},RV,T_{FW}))\xrightarrow{\surd} \\ (\mathit{Recruit}(n,d)\times \mathit{RecruitingForager}(n,d,i_{n,d},RV,T_{FW}+1,dt)),\ \surd\epsilon\{\ \surd\ \} \end{array}}\,(\text{Par1})}{\begin{array}{l} (\mathit{Packer}(n,d,wt)\times \mathit{ForagerAtHive}(n,d,i_{n,d},RV,T_{FW}))|\{\ \surd\ \}\xrightarrow{\surd} \\ (\mathit{Recruit}(n,d)\times \mathit{RecruitingForager}(n,d,i_{n,d},RV,T_{FW}+1,dt))|\{\ \surd\ \} \end{array}}\,(\text{Res1})$$

All hypothesis for applying the rule (Par1) are true so the rule can be applied and then one can apply the rule (Res1) to come to the final conclusion. In this example one can

[4] A proof tree is read from the bottom to the top. At the bottom is the conclusion which describes a behaviour of an agent that should be proved (like in equation A.4). Each hypothesis must be proven by applying rules finally leading to a rule with no hypothesis.

understand how the agents work together, which is at first they have to act at the same time ($\times$) :
The rule (Par1) says that the actions of the two agents are performed sequentially so that they can influence each other:
$searchForager_{(n,d)}\overline{searchForager_{(n,d)}} = \surd$. And at last they have to communicate what is managed by rule(Res1).

Remarks The above example points out that it is important to choose the names of the actions carefully in order to determine which pair of agents is allowed to communicate with each other. In the situation above *only* foragers and packers at the *same node* and with the *same destination* ((n, d)) can communicate.

Weights Weights are priorities and relative frequencies. In the BeeAdHoc MAS no probabilities are explicitly used, therefore the use of relative frequencies is not explained in detail. To enable priorities the Θ operator is to be used.

Example In BeeAdHoc foragers prefer packer instead of swarms. So if both a *Packer* and a *SwarmRecruiter* search for an forager at the same time, the forager always should recruit the packer:[5]

$$\begin{aligned}&\Theta((Packer(n,d,wt) \times SwarmRecruiter(n,d,wt) \times ForagerAtHive(n,d,i_{n,d},RV,T_{FW}))|\{\surd\})\\ &\xrightarrow{\surd}\\ &\Theta((Recruit(n,d) \times SwarmRecruiter(n,d,wt+1) \times RecruitingForager(n,d,i_{n,d},RV,T_{FW}+1)|\{\surd\})\end{aligned} \tag{A.7}$$

At first two of the agents have to communicate. There are only two possibilities :

$$\begin{aligned}&(SwarmRecruiter(n,d,wt) \times ForagerAtHive(n,d,i_{n,d},RV,T_{FW})) \mid \{\surd\} \xrightarrow{\surd}\\ &(Deleted \times SwarmMember(n,d,i_{n,d},RV)) \mid \{\surd\}\end{aligned} \tag{A.8}$$

or

$$\begin{aligned}&(Packer(n,d,wt) \times ForagerAtHive(n,d,i_{n,d},RV,T_{FW})) \mid \{\surd\} \xrightarrow{\surd}\\ &(Recruit(n,d) \times RecruitingForager(n,d,i_{n,d},RV,T_{FW}+1,dt)) \mid \{\surd\}\end{aligned} \tag{A.9}$$

The remaining agents have to wait, which is a $\surd$ action in WSCCS. Other actions are not permitted because of $\mid \{\surd\}$.

[5] The description of the packer agent is in equation 4.9 on page 28, and that of the forager agent is in equation 4.1 on page 23 and that of the swarm recruiter agent is in equation 4.12 on page 29

Which priorities do the two possible interactions have ?
In the first case the rule (Par2) leads to ω^2 and in the second case to ω^4. The Θ operator makes the second case be chosen because of rule (Pri2) that actually is that the packer is recruited by the forager.

A.13.3. The probability workbench

Setting up the workbench The workbench can be compiled at runtime by

use ”directory/rwb.txt”;

in the New Jersey's *sml* shell. The path in directory is relative to the directory of the sml /bin directory. Before compiling, the paths in rwb.txt should be replaced by the appropriate paths.

Usage The workbench offers several features which are listed in the commands below:

rf "filename(.prob/.pro)"	reads the specified input file of the workbench and parses the agents ! Note: an agent can only be parsed once again after recompiling the workbench !
build "name"	builds the transition graph of a parallel (!) agent with the label "name", sequential agents are not accepted
sim "name"	passes through a transition graph of a parallel or sequential agent labelled by "name" *with user interaction* ! Note: graphs of parallel agents require to be built by *build* "name" first
simN "name" n	simulates the agent labelled by "name" *n* steps using the first possible transition ! An addition of the BeeAdHoc version of the workbench
sim_with_build "name"	passes through the transition graph of the agent "name" building the states on demand and hence not requiring a *build* in advance ! An addition of the BeeAdHoc version of the workbench
simN_with_build "name" n	a mixture of *simN* and *sim_with_build*
dupo "filename"	writes the simulation output into "filename" ! might only work in the BeeAdHoc version
co	closes the file opened by *dupo*

If the transition graph is built completely then it should be ensured that a stable state is always reached else the "build" command does not terminate. One might define an agent which terminates all other agents after a finite number of steps. In the BeeAdHoc version of the workbench all *Deleted* agents are deleted during a simulation or the build of a transition graph.

Input files The workbench requires the input to be in a strictly syntactically correct (taking care of some blanks, too). This might differ on some platforms or versions of *sml*. The syntax is slightly different from WSCCS and one has to take care of some special rules:

$$\begin{array}{ll}
\textit{term} ::= & \text{bs E } \textit{seqAgent} \mid \text{bpa E } \textit{parAgent} \mid \text{btr E } \textit{parAgent}/\text{L} \mid \\
 & \text{bpc E } \textit{parAgent}/\text{L} \mid \text{actions } \textit{set} \mid \text{basi L } \textit{set} \\
\textit{set} ::= & \text{a} \mid \text{a,}\textit{set} \\
\textit{seqAgent} ::= & \text{h@ w:}\textit{act}\text{.E} \mid \textit{seqAgent} + \textit{seqAgent} \mid \textit{seqAgent}\backslash/\text{n} + \textit{seqAgent} \\
\textit{parAgent} ::= & (S \mid \textit{parAgent}) \\
\textit{act} ::= & \text{a} \mid \text{a} \wedge -k \mid \text{a} \wedge k \mid \textit{act\#act}
\end{array}$$

where *a* is a label of an action, E of an agent and S of a *sequential* agent. "/n" notes the line break. A relative frequency is noted by *h* and must be replaced by a number as well as the priority *w*. In the notation of an action (act) $\wedge$ is the power operator and *k* a number. The declaration of a sequential agent is introduced by *bs* and of a parallel one by *bpa*. All labels of actions used in the file have to be declared after *actions* and with *basi* one can define a set of permitted actions. To confine a *parallel* agent to these sets, one has to use the declaration *bpc* or *btr*. The latter one takes care of priorities. An example for the syntax can be seen in the following agent system from an input file for the workbench:

```
actions t,finish_1,searchForager_1_2
bs Deleted 1.t:Deleted

bs Packer_1_2_3 1@1.searchForager_1_2^1:Recruit_1_2 + 1.t:Deleted
bs Packer_1_2_2 1@1.searchForager_1_2^1:Recruit_1_2 + 1.t:Packer_1_2_3
bs Packer_1_2_1 1@1.searchForager_1_2^1:Recruit_1_2 + 1.t:Packer_1_2_2
bs Packer_1_2_0 1@1.searchForager_1_2^1:Recruit_1_2 + 1.t:Packer_1_2_1

bs Recruit_1_2 1.t:Recruit_1_2

bs PS_1_2_2 1@1.finish_1^-1:Deleted\
+ 1.t:PackerSender_1_2_2
bpa PackerSender_1_2_2 Packer_1_2_0—Packer_1_2_0—PS_1_2_4

bs Abort_1_2_0 1@1.finish_1^1:Abort_1_2_0\
+ 1.t:Abort_1_2_0
bs Abort_1_2_1 1.t:Abort_1_2_0
bs Abort_1_2_2 1.t:Abort_1_2_1
```

basi L t

btr TEST1 Abort_1_2_5—PackerSender_1_2_3/L

Bibliography

[AC03] A.A.Pirzada and C.McDonald. A review of secure routing protocols for ad hoc mobile wireless networks. 2003. pp 118-123.

[aHFLS02] Yi an Huang, Wei Fan, Wanke Lee, and Philip S.Yu. Cross-feature analysis for detecting ad-hoc routing anomalies. 2002.

[aHL03] Yi an Huang and Wanke Lee. A cooperative intrusion detection system for ad hoc networks. 2003.

[CJ] T. Clausen and P. Jacquet. Rfc3626. http://ietfreport.isoc.org/idref/rfc3626/.

[CS93] S. Camazine and J. Sneyd. A model of collective nectar source selection by honey bees: self-organization through simple rules. Journal of Theoretical Biol. 149 pp 547-571, 1993.

[Das98] Dipankar Dasgupta. *ARTIFICIAL IMMUNE SYSTEMS AND THEIR APPLICATIONS*. Nov 1998.

[dC] Leandro Lune de Castro.

[dCvZ99] de Castroand and von Zuben. *Artificial Immune Systems Part 1, Basic Theory and Applications*, Jan 1999.

[dCvZ00] de Castro and von Zuben. *Artificial Immune Systems Part 2, A Survey of Applications*, 2000.

[Dik] Jeff Dike. User-mode linux, homepage: http://user-mode-linux.sourceforge.net/.

[Dil] Prof. Dr. Werner Dilger. Vorlesung kueunstliche immunsysteme, technische universitaet chemnitz, sommersemester 2004.

[ES03] Prof. Dr. Claudia Eckert and Thomas Stibor. Grundlagen des immunsystems und anomalieerkennung durch kueunstliche immunsysteme, November 2003.

[Fee99] Laura Marie Feeney. Investigating the energy consumption of an ieee 802.11 network interface. Technical report, Swedish Institute of Computer Science, December 1999.

[Fri65] Karl von Frisch. *Tanzsprache und Orientierung der Bienen.* Springer Verlag Berlin, Heidelberg, 1965.

[HB04] Yih-Chun Hu and David B.Johnson. Securing quality-of-service route discovery in on-demand routing for ad hoc networks. *SASN 04*, 2004.

[HPB02] Yih-Chun Hu, Adrian Perrig, and David B.Johnson. Ariadne:a sceure on demand routing protocol for ad hoc networks. *MobiCom 02*, 2002.

[HPB03] Yih-Chun Hu, Adrian Perrig, and David B.Johnson. Rushing attacks and defense in wireless ad hoc network routing protocols. *WiSe 2003*, 2003.

[Imm] *Immune System.*

[JM01] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors, August 2001.

[Ne04] N.Milanovic and M. Malek et.al. Routing and security in mobile ad hoc networks. *IEEE Computer*, Feb 2004. pp 61-65.

[net] *netfilter/iptables.* http://www.netfilter.org/.

[NLTG] Erik Nordstroem, Henrik Lundgren, David Lundbergand Christian Tschudin, and Per Gunningberg. Ad hoc protocol evaluation testbed 0.4, homepage: http://apetestbed.sourceforge.net/.

[Nora] Erik Nordström. Mac-kill. http://apetestbed.sourceforge.net.

[Norb] Erik Nordstroem. Ad-hoc on-demand distance vector routing, homepage: http://core.it.uu.se/adhoc/aodvuuimpl.

[ns2] *The Network Simulator - NS-2.* http://www.isi.edu/nsnam/ns/.

[Pe] P.Brando and S.Sargento et.al. Secure routing in ad hoc networks for reactive next hop routing protocols. pp 109-115.

[Rey02] J. Reynolds. Assigned numbers. RFC 3232, January 2002.

[rfc81] Internet protocol. RFC 791, Information Science Institute, University of Southern Carolina, December 1981.

[RJ98] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors, October 1998.

[SCS91] Thomas D. Seeley, Scott Camazine, and James Sneyd. Collective decision-making in honey bees: how colonies choose among nectar sources. *Behavioral Ecology and Sociobiology*, 28:277–290, 1991.

[See92] Thomas D. Seeley. The tremble dance in the honey bee: message and meanings. *Behavioral Ecology and Sociobiology*, 31:375–383, 1992.

[See94] TD Seeley. Honey bee foragers as sensory units of their colonies. Behav. Ecol. Sociobiol. vol 34 pp 51-62, 1994.

[SG] Alessio Botta Salvatore Guadagno. D-itg, distributed internet traffic generator. http://www.grid.unina.it/software/ITG.

[sil] *INFINEONS Partner Platform for Silicon Based Security Solutions.*

[ST92] TD Seeley and WF Towne. Tactics of dance choice in honey bees: Do foragers compare dances? Behav. Ecol. Sociobiol. vol 30 pp 59-69, 1992.

[Sum00] David J. T. Sumpter. From bee to society: An agent-based investigation of honey bee colonies. *Phd Thesis submitted at the University of Manchester*, march, 2000.

[Tim] Dr. J. Timmis. Aisdef. Department of Computer Science and Department of Electronics University of York.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization, Nov 1994.

[WF05] Horst F. Wedde and Muddassar Farooq. A performance evaluation framework for nature inspired routing algorithms. *Lecture Notes in Computer Science*, Volume 3449:136–146, January 2205.

[WFB$^+$04] Prof. Dr. Horst F. Wedde, MSc Muddassar Farooq, Lars Bensmann, Thomas Buening, Mike Duhm, Rene Jeruschkat, Gero Kathagen, Johannes Meth, Kai Moritz, Christian Mueller, Thorsten Pannenbaecker, Bjoern Vogel, and Rene Zeglin. Beehive an energy-aware scheduling and routing framework. Technical report, Chair 3, University of Dortmund, Germany, September 2004.

[wir05] Pg beeadhoc: Zwischenbericht. Technical report, Universität Dortmund, Fachbericht Informatik, 2005.

[Zap01] M.G Zapata. Secure ad hoc on-demand distance vector routing saodv routing. Oct 2001.