UNIVERSITY OF OSLO
Department of Informatics

Ant-Inspired Approach for Resource Localization in Mobile Ad-Hoc Networks

Master's Thesis

Maren Feragen

May 3, 2010

Abstract

Computer networks are constantly emerging, and in today's world, we see new needs and new kinds of networks addressing these needs. Situations such as emergency and rescue operations present a demand for mobile ad-hoc networks, a kind of networks with fundamentally different characteristics than traditional wired networks. Scarce resources and possibly low connectivity call for new techniques and approaches to traditional problems.

Where possible, resource localization tools will probably be included in the applications needing it, enabling them to tune the search to the applications needs and characteristics. However, such needs are not always possible to predict a priori, and we then need a system enabling nodes to autonomously localize needed resources that are not present in the node itself. In this thesis, we propose a general-purpose resource localization solution for mobile ad-hoc networks.

Nodes in mobile ad-hoc networks should function autonomously with as little outside intervention as possible, preserving the self-* properties. Such systems are common in nature, and may be used as inspiration when designing computer systems. In this thesis, we look at how stigmergy — the foraging behavior of ants — may be used to localize resources in a mobile ad-hoc network. When walking between their nest and a food source, ants leave pheromone trails on the ground, and by probabilistically choosing branches with the most pheromone, they are able to find the shortest path between their nest and the food. This behavior has inspired a lot of solutions to optimization problems like routing.

We have designed and implemented a system exploiting what we call *opposite stigmergy*, where artificial ants are released from a resource-requesting node and at each intermediate node probabilistically choose the *least recently used* neighbor as next hop. This makes ants search every nook and cranny of the network, as opposed to regular ant-inspired algorithms, where ants are concentrated on one or a few most optimal paths.

Our preliminary performance tests show that our ant-inspired approach has, if tuned correctly, the potential to outperform the much simpler flooding-based solution in certain scenarios. The ant solution has shown to be able to localize resources with less sent and received messages if more than one node requests the same (or similar) resources. For scenarios where nodes have a high number of neighbors, however, we see that the ant approach requires a lot of time and resources to localize a resource. We also suggest a few potential fixes that may be able to lower both the response time as well as the resource consumption, enabling the ant solution to perform even better.

Acknowledgements

First and foremost, I want to thank my supervisor, Ellen Munthe-Kaas, for giving me excellent support, guidance, ideas; and not least for introducing me to the somewhat out of the ordinary — but very intriguing — topic for this thesis. I also want to thank Matija Pužar and Piotr Kamisinski for helping me out during the test phase and for patiently assisting me all those times my code or computer didn't do what I wanted it to. Finally, I thank my friends and family for their support and encouragement during the work with this thesis.

Maren Feragen
University of Oslo
May 3, 2010

Contents

1 Introduction **1**

1.1 Background . . . 2

1.2 Motivation . . . 3

1.2.1 Rescue Operations — A Case Study . . . 3

1.2.2 Application Domain . . . 4

1.3 Problem Description . . . 5

1.4 The SIRIUS Project . . . 6

1.5 Outline . . . 6

2 Mobile Ad-Hoc Networks and Autonomic Networking **9**

2.1 Mobile Ad-Hoc Networks . . . 9

2.1.1 Characteristics of Mobile Ad-Hoc Networks . . . 9

2.1.2 Issues in Mobile Ad-Hoc Networks . . . 10

2.1.3 Routing in Mobile Ad-Hoc Networks . . . 12

2.1.4 Applications . . . 16

2.2 Autonomic Networking . . . 17

2.2.1 Autonomic Networks . . . 17

2.2.2 Self-*: Properties of an Autonomic Network . . . 17

2.2.3 Autonomy in MANETs . . . 20

3 A Bug's Life **21**

3.1 Communication . . . 21

3.2 Stigmergy and Ant Colony Optimization . . . 22

3.2.1 Pheromones . . . 23

3.2.2 Ants and Their Pheromones . . . 23

3.2.3 Ant Colony Optimization . . . 25

4 ACO Approaches to Some Traditional Problems **27**

4.1 The Ant Colony Optimization Metaheuristic . . . 27

4.2 The Traveling Salesman Problem . . . 29

4.2.1 Solving the TSP with ACO Algorithms . . . 29

4.2.2 AntSystem . . . 30

4.3 The Routing Problem . . . 31

4.3.1 AntNet — Routing in Traditional Networks 32
4.3.2 AntHocNet — Routing in Mobile Ad-Hoc Networks . 35

5 Design 39
5.1 Goal 39
5.2 Assumptions 39
5.3 Requirements 40
5.4 General Idea 40
5.5 Issues 42
5.5.1 Scheduling 42
5.5.2 Communication 43
5.5.3 Supported Network Topology 45
5.5.4 Resources 45
5.5.5 Pheromone Traces 46
5.5.6 Ants 48
5.6 The Localization Algorithm 52

6 Implementation 55
6.1 Remarks 55
6.2 Programming Language 56
6.3 Developing for NEMAN 56
6.4 The sockaddr_in structure 56
6.5 Logging 57
6.6 Neighbor Information 58
6.6.1 Topology Information Retrieval 58
6.6.2 Neighbor Registry 59
6.6.3 Topology Changes During Resource Localization . . . 59
6.7 Resources 61
6.7.1 Local Resource Information 61
6.7.2 Resource Sharing 61
6.7.3 Resource Location Info 62
6.8 Pheromone Traces 63
6.8.1 Pheromone Data Structure 63
6.8.2 Pheromone Initialization 65
6.8.3 Pheromone Updates 65
6.9 Ants and Ant Memory 65
6.9.1 Choosing the Next Hop 66
6.9.2 Loop Elimination 68
6.9.3 Ant Communication 68
6.10 Resource Localization 68
6.10.1 Search Initiation 68
6.10.2 Search Termination 69
6.11 Utilities 69
6.12 Program Flow 70

6.12.1 Threads 70

7 Test Setup 71

7.1 Evaluation Techniques 71

7.2 Testing Environment 73

7.2.1 Simulation vs Emulation 73

7.2.2 ns-2 73

7.2.3 NEMAN 73

7.2.4 OLSR daemon 74

7.3 Emulation and Analysis Tools 74

7.3.1 setdest 74

7.3.2 tcpdump 75

7.4 A Flooding Solution 75

7.4.1 Issues with Flooding 75

7.4.2 Flooded Requests 76

7.5 Monitoring 77

7.6 Metrics 77

7.6.1 Response Time 78

7.6.2 Resource Usage and Utilization 80

7.7 Test Scenarios 83

7.7.1 Chain Scenario 83

7.7.2 Grid Scenario 83

7.7.3 Mobility Scenario 84

7.8 Test Scenario Implementation 85

7.8.1 Static Scenarios 86

7.8.2 Mobility Scenario 86

7.9 Workload 87

7.9.1 Parameters 89

7.9.2 Scenario Properties 89

7.9.3 Single Localization 90

7.9.4 Location Learning 91

8 Performance Evaluation 95

8.1 Influencing Factors 95

8.1.1 Topology Initialization and Updates 95

8.1.2 Time Inaccuracy 96

8.1.3 Average Neighborhood Size 96

8.2 Results — Response Time 97

8.2.1 Single Localization 97

8.2.2 Location Learning 100

8.2.3 Conclusion 102

8.3 Results — Bandwidth Usage 103

8.3.1 Single Localization 103

8.3.2 Location Learning 105

8.3.3 Conclusion . . . 107
8.4 Results — Processing Power Usage . . . 108
8.4.1 Single Localization . . . 109
8.4.2 Location Learning . . . 109
8.4.3 Conclusion . . . 110
8.5 Results — Processing Power Utilization . . . 110
8.5.1 Single Localization . . . 111
8.5.2 Location Learning . . . 111
8.5.3 Conclusion . . . 112

9 Conclusion and Further Work 113
9.1 Contribution . . . 113
9.2 Performance Evaluation . . . 114
9.3 Critical Assessments . . . 115
9.4 Further Work . . . 116
9.4.1 Resource Localization in Sparse Networks . . . 116
9.4.2 Resource Goodness . . . 117
9.4.3 Further Implementation, Testing and Analysis . . . 118

Bibliography 120

Appendices 127

A A Sample Scenario File 127
A.1 Chain Scenario . . . 127

B The Flooding Solution 129
B.1 Data structures . . . 129
B.1.1 Logging . . . 130
B.1.2 Program Flow . . . 130

C Source Code 131
C.1 Program Layout — Ant Solution . . . 131
C.2 Program Layout — Flooding Solution . . . 131
C.3 Building the Source Code . . . 132
C.4 Running the Source Code . . . 132

D Code Examples 133
D.1 send_forward_ant() . . . 133
D.2 receive_ants() . . . 134
D.3 handle_received_ant() . . . 135
D.4 handle_forward_ant() . . . 136
D.5 handle_backward_ant() . . . 136

E CD Contents **139**
E.1 /implementation 139
E.1.1 /implementation/ant_solution 139
E.1.2 /implementation/flooding_solution 139
E.2 /test_setup 139
E.2.1 /test_setup/scenarios 140
E.3 /analysis 140

List of Figures

2.1 A simple MANET 10
2.2 Sample MANET before and after partitioning 11
2.3 Message forwarding in LSR versus OLSR 14
2.4 Epidemic routing in a MANET with two partitions 15

3.1 Two kinds of trail-leaving ants 24
3.2 Ant colony is unable to converge to shortest path 25

4.1 Pseudo-code for the ACO metaheuristic 29

5.1 The disadvantage of backward ants depositing pheromones . . 47
5.2 Allowing ant loops increases network utilization 51
5.3 Pseudo-code for the ant solution 53

6.1 C code for binding a socket to a specific device 57
6.2 sockaddr_in structure 57
6.3 Neighbor structure 59
6.4 Backward ant — neighbor lost 60
6.5 Forward ant — neighbor lost 60
6.6 A sample resource file 61
6.7 Resource structure 62
6.8 Resource info structure 62
6.9 Two-dimensional linked list structure 64
6.10 Pheromone structure 64
6.11 Ant structure 66
6.12 Neighbor probability structure 67
6.13 Loop elimination 68
6.14 Doubly linked list structure 70

7.1 Response time metric 78
7.2 Different interpretations of the response time metric 78
7.3 Chain topology 83
7.4 Grid topology 84
7.5 Mobility topology 85

7.6 Chain topology NEMAN screenshot 86
7.7 Grid topology NEMAN screenshot 87
7.8 Mobility topology NEMAN screenshots 88
7.9 Chain topology with one requesting node 90
7.10 Grid topology with one requesting node 91
7.11 Mobility topology with one requesting node 92
7.12 Grid topology with three requesting nodes 93
7.13 Mobility topology with five requesting nodes 94

8.1 Number of neighbors per second in the mobility scenarios . . 98
8.2 Response time in grid scenario with location learning 101
8.3 Response time in M-dense scenario with location learning . . 101
8.4 Response time in M-sparse scenario with location learning . . 102
8.5 Message sizes for flooding and ant solutions 105
8.6 Utilization in each single location scenario 110
8.7 Utilization in each location learning scenario 111

B.1 Flood_package structure 129
B.2 Node_seq structure 130
B.3 Request_answer structure 130

List of Tables

8.1 Neighbors in the mobility scenarios 98
8.2 Number of hops used in each scenario 99
8.3 Time spent in each scenario 99
8.4 Messages sent and received in single localization scenarios . . 104
8.5 Bytes sent and received in single localization scenarios 104
8.6 Messages sent and received in location learning scenarios . . . 106
8.7 Bytes sent and received in location learning scenarios 106

List of Abbreviations

ACO	Ant Colony Optimization
AODV	Ad Hoc On Demand Distance Vector
AS	AntSystem
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
CTS	Clear To Send
DCF	Distributed Coordination Function
DMMS	Distributed Multimedia Systems
IEEE	Institute of Electrical and Electronics Engineers
LSR	Link-State Routing
MANET	Mobile Ad-Hoc Network
MPR	Multi-Point Relay
OLSR	Optimized Link State Routing
RREP	Route Reply
RREQ	Route Request
RTS	Ready To Send
SIRIUS	Sensing, Adapting and Protecting Pervasive Information Spaces
TCP	Transmission Control Protocol
TSP	Traveling Salesman Problem
UPD	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

"I'm not afraid of computers taking over the world. They're just sitting there. I can hit them with a two by four."

— *Thom Yorke*

In mobile ad-hoc networks, nodes should function autonomously, and they should be able to adapt to their environment and any changes in it without any external intervention. For nodes to be able to adapt to their environment, they obviously need to have a certain knowledge about their environment and to somehow be aware of any changes within this environment.

The environment of a node in a mobile ad-hoc network is made up of the nodes in the network, and how these are placed, their mobility, speed and any other characteristics that a node may enclose, such as what resources are present at which nodes in the network. One kind of environmental changes is thus changes in the network topology, which may occur frequently because of node mobility. Another possible environmental change is changes to the resource situation at one node.

As we are dealing with mobile ad-hoc networks, it is important that nodes are able to monitor their environment and perform any required adaptations without any external intervention — they need to function autonomously. Autonomous, self-adapting systems are common in nature, and have been used as inspiration for solutions to a lot of computer-related problems, especially optimization problems. The most common source of inspiration is ants and their foraging behavior. A lot of research has been done on ant-inspired approaches to optimization problems like the routing problem, both in traditional, wired networks as well as in mobile ad-hoc networks.

With this thesis, we want to look at how ants and their behavior may be used as inspiration for other kinds of problems, and to find out if such approaches may be feasible also in other scenarios than the typical optimization

problems. As our application domain we have chosen *resource localization* in mobile ad-hoc networks. The purpose of the resulting solution is to enable nodes to search for any resource at any time without the need for any prior knowledge about which resources will be requested or when requests may be issued beforehand. This way, nodes may issue searches for a given resource whenever they discover a need for knowledge about the resource situation within the network.

1.1 Background

Over the last few decades, there has been a shift from the more traditional way of networking, with a static, wired infrastructure connecting relatively few and homogeneous nodes, to a more widespread, dynamic and thus more complex infrastructure. We now see large networks, a mix of wired and wireless infrastructure, connecting a large number of more heterogeneous nodes. However, we also see a need for networks in places that networking has before been unthinkable, as there exists no infrastructure, nor, perhaps, any plans of ever making one. Examples of places like this may be sparsely populated, desolate regions and third world countries, but also in more urban settings like in tunnels, where outside signals would have a small chance of reaching in.

For this purpose, a new kind of networks, called *Mobile Ad-Hoc Networks* (MANETs) have been introduced. A MANET is an autonomous network of mobile, wireless nodes. In a MANET, there is no fixed infrastructure, and all nodes in the MANET thus need to be able to work both as an end system and a router, in order to make it possible for nodes not within range of each other to communicate with each other via other nodes in the MANET.

In situations like emergency and rescue operations, it is of great importance to set up a well functioning network as fast as possible, in order to be able to exchange information about the emergency and potential victims between the rescue workers as fast as possible. Because no fixed infrastructure is needed for establishing MANETs, they can quickly be established, and are thus thought of as a good solution in such situations.

Nodes in a MANET may be highly mobile, and the topology within the network may thus change rapidly, as nodes move in and out of range of each other. Also, nodes in a MANET typically have scarce resources, such as bandwidth, battery power and CPU. These characteristics put harder requirements on the algorithms used to maintain this kind of networks.

Typical for the more traditional networks is a human network administrator running the network, making sure everything works as desired, doing the necessary adjustments and repairs. As networks are growing larger and more complex, along with the introduction of MANETs, the workload on the human administrator increases or becomes unfeasible, as there might

not be an obvious candidate for the network administrator role. It is also getting harder and harder to anticipate when implementing systems what optimizations and adaptations will be needed at runtime. Thus, with today's new networking world, it is desired to automate these processes: To let the network itself do the necessary adaptation and optimization, to make the necessary changes on itself when needed, to protect itself from threats, and to repair itself when needed.

1.2 Motivation

As mentioned above, MANETs may be used where there is no fixed infrastructure. The lack of infrastructure may be due to remote location, and thus few or no possible users, restricted economy and thus the power to build the needed infrastructure, but also unforeseen incidents such as a fire, an earthquake or some other natural disaster.

In many of these settings, such as in rural locations, networking is not a prerequisite. However, situations may arise where being able to communicate efficiently would be highly advantageous. Examples of such situations are emergency operations, both in rural locations as well as more urban locations such as tunnels, and during battlefield operations. MANETs may also be used in less critical situations, for example in wildlife tracking [19].

1.2.1 Rescue Operations — A Case Study

In April 1998, the personnel on a metro train with several hundred persons on board discovered a fire in Majorstutunnelen, a 1790 metre long metro tunnel in the center of Oslo [12]. The personnel tried contacting their supervisors, but with no luck, and let their passengers off inside the tunnel, hoping they would get out safely without supervision. At the same time, there was also another train inside the same tunnel. As no one was able to contact anyone on the outside, the personnel in the other train knew nothing about the fire until they saw people walking towards them along the tracks inside the tunnel. Luckily, the driver was able to stop the train, avoiding to hit any of the passengers from the other train.

In such a scenario, the ability to communicate, both with someone on the outside as well as with the personnel on board other trains nearby, is crucial. In our scenario, luckily, no one was physically hurt, and everyone managed to get out of the tunnel. The fire turned out to be small, and was shortly after extinguished by fire personnel. However, it is not hard to imagine a less happy ending: The fire could have been worse, the driver of the other train could have been unable to stop his train in time, people could have been hurt by the electric current in the railway tracks. In such a scenario, the amount of rescue personnel involved in the rescue operation might get very high, as both fire fighters, police personnel and medical personnel may

need to enter the tunnel to contribute. With such an amount of personnel in action, the need for efficient communication is even greater: Who is doing what, how many victims are located where, who needs help first, who has got what equipment where, and so on.

If all rescue workers carried with them a small device with the ability to communicate with other devices, they would be able to set up a MANET without much effort, enabling them to exchange messages, pictures, maps, resource information and all other kinds of information useful to the rescue operation. They might also place sensors on their patients, enabling them to monitor moderately hurt people without being physically close to them. The sensors could trigger an alarm if the patient's condition gets worse, telling medical personnel to go check up on the patient.

Other sensors and cameras might monitor the tunnel environment with respect to for example temperature and the amount of poisonous gases in the air inside the tunnel, making sure it is safe to continue the rescue operation inside the tunnel, again giving alerts to the appropriate personnel whenever the conditions get worse.

Many of these tasks could be accomplished with the help from existing systems or even plain communication between the rescue personnel. However, one rescue worker may carry certain kinds of equipment or even knowledge that other workers may benefit from. If the resource holder could answer to resource requests without actually having to pick up his or her device and type an answer, this rescue worker would be free to work with other tasks for more of the time, making this worker more efficient, possibly saving more lives or in some other way minimizing damage. Thus, we would like a system where one could send a message "I need resource X", and the system would locate this resource within the MANET without any human intervention.

1.2.2 Application Domain

A rescue operation is of course not the only domain where such a system might be helpful. Any system where one node might need resources not present within the node itself would benefit from such a system. However, there are a few characteristics that are common for the kind of application domains the work in this thesis is targeted at. These characteristics put some restrictions on the solutions developed:

- As resources in a MANET are typically scarce, we would like a solution with a high utilization of the present resources: We want as little waste as possible of both bandwidth as well as processing power.

- MANET node characteristics include frequent joins and leaves as well as partitioning.

- No other system for sharing and dissemination resource information

is already present within the system, or the existing solution is not adequate.

The actual consequences of the first two restrictions will be further discussed in later sections, but we do already state that these restrictions severely complicate both the design and implementation of systems targeted at MANETs.

If it is possible to predict beforehand what kind of resources might be present in the network and what resources may be requested, resource dissemination and localization tools will probably be included in the system, and may then be tailored to fit the needs and characteristics of the system in question. However, it is not always possible to foresee the exact needs and development of a system. Thus, what we aim to develop in this thesis is a solution for use either when no such tools are available, or as an addition to any existing tools. Thus, we try to develop a general purpose tool that may be used to look for any resource at any time in any MANET.

1.3 Problem Description

A lot of research has been done on how well biologically inspired algorithms apply to computer network problems, both in traditional networks as well as MANETs. These approaches look at how we can make computer systems simulate behavior seen in the nature. Self-organizing systems exist naturally in nature, where they function without any external interference or central control. They adapt to changes around them, making them more robust to environmental changes and increasing their own survivability. One example of such biological systems is insects, like ant colonies, termites or bees, who communicate through *stigmergy*. The term stigmergy was introduced by Grassé in 1959 [26]. Grassé defined stigmergy as

> "Stimulation of workers by the performance they have achieved."
> Dorigo et al. [9]

In other words, stigmergy describes insects' indirect communication mediated by changes to the environment. This phenomenon is also referred to as *swarm intelligence*. In the case of termites and ants, stigmergy is ensured by depositing the chemical substance *pheromone* in the environment [36].

Another example is *autopoiesis*, or *self-production*, which refers to systems of components that are able to reproduce themselves, and in this way self-maintain the system [36]. Other such examples exist, such as *decrease of entropy* [36], and these natural systems may all be used as inspiration when designing autonomous computer systems and networks.

In this thesis, we look at some of the algorithms from the *Ant Colony Optimization* [10] research field, and try to map these techniques to our problem: Locating a given resource within a MANET. More specifically, we

will look at the behavior of ants during their search for food, and try to exploit the characteristics of their movement in our own search for resources in a MANET.

1.4 The SIRIUS Project

This thesis is written as part of the SIRIUS (Sensing, Adapting and Protecting Pervasive Information Spaces) project at the Distributed Multimedia Systems (DMMS) research group. The project mainly focuses on three challenges:

- *Sensing/monitoring* as a high-level service for applications.
- *Adaptation* through a framework supporting autonomous adjustments of all objects in such systems, minimizing unwanted side effects from individual adaptations.
- *Protection* through a tool for identifying and detecting normal as well as abnormal behavior in the system, perform analysis and carry out counter efforts demanding a minimal amount of manual intervention.

The project aims to provide concepts and mechanisms for system developers, enabling them to create large-scale pervasive systems. In addition to this, a platform shielding the application developer from the underlying complexities is needed, as one cannot expect every developer to understand and address all issues introduced by the new turn in networking.

This thesis is a contribution to the second point above; *adaptation*. Resource localization is to be done autonomously, and nodes holding and seeking resource information need to be able to adapt to any changes in their environment, such as node mobility, which may lead to resource-holding nodes moving or even leaving the MANET. Also, by gathering resource information, nodes may get information about the status of the network, and may thus be able to adapt to any changes without any outside intervention.

1.5 Outline

The rest of this thesis is organized as follows: Chapter 2 provides a thorough explanation of Mobile Ad-Hoc Networks and autonomic networking. In Chapter 3, we look at ants and stigmergy, the biological phenomenon that is the inspiration of our resource localization system. Chapter 4 takes a deeper look at how ant behavior may be exploited to make solutions to some traditional computer network problems.

Our resource localization design is explained in Chapter 5, whereas the more technical details of our implementation are provided in Chapter 6.

The test setup used in our experiments, along with a presentation of some of the most important tools used during the testing, is provided in Chapter 7. The test results and system evaluation follows in Chapter 8. At last, we conclude and present further work in Chapter 9.

The thesis also includes four appendices. Appendix A contains a sample NEMAN scenario file with additional comments and explanations. In Appendix B, we explain some of the details of the flooding solution used for comparison during the performance analysis, whereas Appendix C provides an overview of the source code of both the ant solution as well as the flooding solution. Appendix D lists some of the most important parts of the source code for our application. The final appendix, Appendix E explains the content of the CD appended to this thesis.

Chapter 2

Mobile Ad-Hoc Networks and Autonomic Networking

> *"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."*
>
> — *Andrew S. Tanenbaum*

In this chapter, we will go through a few topics relevant for the understanding of the rest of the thesis. First, we provide a detailed explanation of mobile ad-hoc networks, some issues, their applications and a few relevant routing protocols. Then we also give an overview on autonomic networking and self-* systems.

2.1 Mobile Ad-Hoc Networks

2.1.1 Characteristics of Mobile Ad-Hoc Networks

A Mobile Ad-Hoc Network (MANET) is a collection of mobile, autonomous nodes, together forming a network without using any fixed infrastructure and without any outside intervention [13, 38]. Typically, there is no centralized control, and the network is thus dependent of node cooperation to function properly.

Because of the lack of fixed infrastructure, each node is acting as both an end node as well as a router, cooperating on the task of getting each message from source to destination. This means that every node needs to run the (same) routing protocol, which puts a bit of extra load on the nodes.

Each node will have a *neighborhood*, meaning a set of *neighbors*, which are the nodes that are within range of the node and thus may be reached with *direct* communication. All other nodes are outside range, and may thus only be reached with the help from *indirect* communication, meaning that

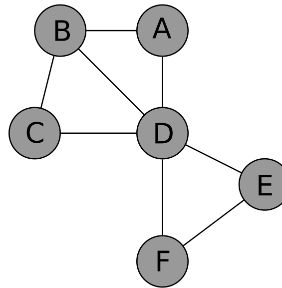


Figure 2.1: A simple MANET.

all communication needs to happen via one or more other nodes within the MANET. This is shown in Figure 2.1, where node A has two neighbors which may be reached directly, namely nodes B and D, whereas node C may be reached either via node B or node D.

2.1.2 Issues in Mobile Ad-Hoc Networks

Because of the characteristics of MANETs, some issues that we do not usually see in regular networks arise.

Node Heterogeneity

Nodes in a MANET may vary from regular laptops to small PDAs and even smaller sensing devices. Thus, the resources available in a MANET may be scarce: Small devices typically have less resources, such as CPU and memory. Also, the transmission range may vary from device to device, possibly leading to asymmetric connectivity.

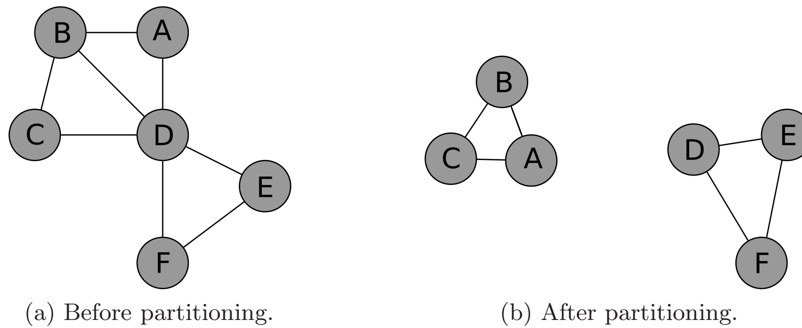
Mobility

Nodes in a MANET are typically mobile. As nodes are moving, they need to run on battery power. Mobile nodes may be placed anywhere, and their movements may not be human controllable. Thus, we do not want to have to change their batteries every day or even every week or month; We want their batteries to last as long as possible.

Higher mobility also leads to more frequent topology updates, as connectivity between nodes changes, leading to more load on the network as topology updates need to be disseminated throughout the network.

Partitioning

MANETs may also be classified as either *sparse* or *dense*. The density of a MANET is determined by measuring the average number of neighboring nodes in the MANET [25]. A sparse MANET is a MANET where each node



(a) Before partitioning. (b) After partitioning.

Figure 2.2: Sample MANET before and after partitioning

has few neighbors and thus low connectivity, typically because of few nodes per area unit, whereas a dense MANET is a MANET where nodes have a high number of neighboring nodes and thus a high connectivity.

Partitions may occur in the MANET if there are two or more groups of nodes with no link between them, i.e. none of the nodes in one group is within range of any of the nodes in the other group. The more sparse a MANET is, the bigger the probability of partitioning.

When a network is partitioned, no communication is possible between nodes in different partitions, as it is impossible to establish a path between the two nodes. This is shown in Figure 2.2. In Figure 2.2a, we see a MANET consisting of only one partition, and we thus have a path between nodes A and F. After partitioning, shown in Figure 2.2b, the large partition has been split into two smaller partitions, one containing node A and the other containing node F, thus we no longer have a path between nodes A and F.

By allowing delivery to take an arbitrary large amount of time, this problem may be solved by using special routing techniques. These will be further explained in Section 2.1.3.

Wireless Communication

Communication within a MANET is based on *wireless communication* using the IEEE 802.11 standard. With wireless transmission comes a few issues, one of which is *collisions*. If several nodes within the same area transmits a message at the same time, a collision will occur. The more nodes and thus traffic in the MANET, the higher the probability of packets colliding. Colliding packets cause nodes to retransmit, which implies a higher load on both the nodes themselves as well as the network.

The basic access method for IEEE 802.11 is the *Distributed Coordination Function* (DCF) [45]. The DCF uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). In this scheme, all nodes ready to send a message listens on the channel to hear if anyone else is transmitting, a technique also referred to as *physical carrier sense*. If the channel is free,

the node may send. If not, the node must wait until it is, and then enter a random back off procedure where it waits for a random period of time. The random back off procedure is used to make sure several nodes don't start transmitting at the same time immediately after the previous transmission was done. Because of this, however, in a dense network with a large amount of transmissions, nodes will often have to wait before they may send messages.

However, this approach assumes that all nodes are within range of each other, a rather unrealistic assumption. To solve this problem, often referred to as the *hidden node problem*, a carrier sense mechanism called *virtual carrier sense* is also used. In order to avoid collisions, a node wanting to send something must reserve the medium for a specified amount of time by transmitting a *ready to send* frame. If the destination node is ready to receive, it replies with a *clear to send* frame. All nodes within range of either sender or receiver will hear at least one of these two, and will thus know that the medium is occupied for the reserved time period.

This procedure makes the probability of a collision smaller, as collisions will only happen if two nodes decide to send a message at the exact same point in time. However, it might also make nodes wait quite some time before they get to send their messages.

2.1.3 Routing in Mobile Ad-Hoc Networks

Nodes in a MANET may move rapidly and unpredictably. This, together with the restricted resources within a MANET, puts higher restrictions on a MANET routing protocol than on a regular routing protocol designed for traditional networks:

- The protocol must not use too much bandwidth on route establishment and maintenance.

- The protocol must adapt fast to topology changes as well as traffic and propagation changes.

A wide extent of protocols for routing in MANETs have been developed. These may be classified according to several criteria [13]:

- *Communication model:* Decides if the protocol is designed for single channel or multi-channel communication. Multi-channel protocols combine channel assignment and routing functionality, whereas in single channel protocols, nodes communicate over the same logical wireless channel. This is the most common communication model.

- *Structure:* Distinguishes between uniform protocols, where all nodes play an equal role in the network, and non-uniform protocols, where

only some nodes participate in route computation. As less nodes participate in route computation, non-uniform protocols may achieve better scalability than uniform protocols.

- *State information:* Distinguishes between topology-based and destination-based protocols. In topology-based protocols, which include link state protocols, all participating nodes maintain large-scale topology information. In destination-based protocols, on the other hand, nodes maintain only some local topology information. This class of protocols includes distance-vector protocols, in which nodes maintain distance and next hop information for all destinations.

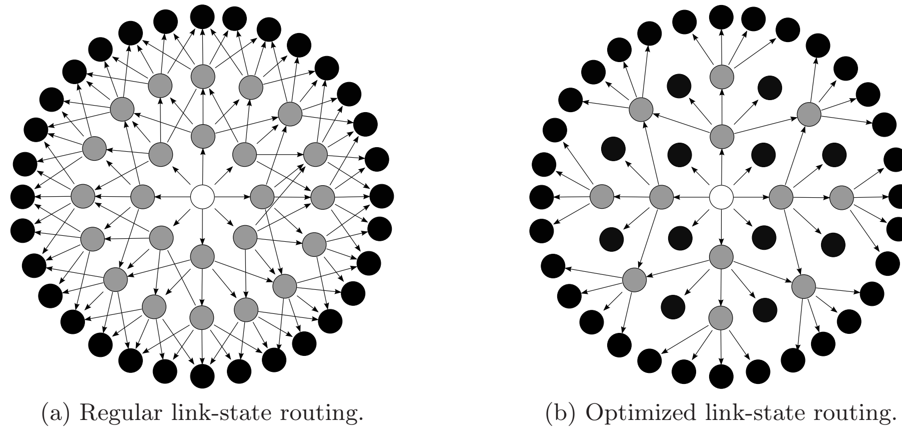
- *Scheduling:* Routing protocols may be either proactive or reactive. A proactive protocol keeps routing information for all destinations at all times, whereas reactive protocols only compute routes for a destination upon request. Proactive protocols thus minimize delay in obtaining a route, but may use significantly more network resources, as routes may be computed for destinations that no message will ever be destined for. The dissemination of route requests in on-demand protocols, on the other hand, requires a significant amount of flooding, which puts heavy load on the network. Also, as nearby nodes are bound to rebroadcast a message more or less at the same time, there is a great chance of contention and collisions occurring.

In addition to this, MANETs may also, as mentioned in Section 2.1.2, be classified as either sparse or dense. Routing in dense MANETs is more similar to traditional routing than in sparse networks, as network partitioning and merging are fairly infrequent events and thus there is one only partition that needs to be considered. With sparse networks and several partitions, however, there is an additional problem regarding how to send messages from one partition to another and how to manage partition merging. In the following we will first go through a couple of routing protocols for dense networks, before we give a few examples on routing in sparse MANETs.

Routing in Dense MANETs

Optimized Link-State Routing *Optimized Link State Routing* (OLSR) is a *non-uniform*, *neighbor selection based*, *proactive* routing protocol, meaning that not all nodes participate in the route computation, and that only information about a node's neighbors is disseminated throughout the network [13].

OLSR is, as the name implies, an optimized version of *link-state routing* (LSR), one of the main classes of routing protocols in traditional wired, packet-switched networks. In regular link-state routing, link state information from each node is distributed to all other nodes in the network upon



(a) Regular link-state routing. (b) Optimized link-state routing.

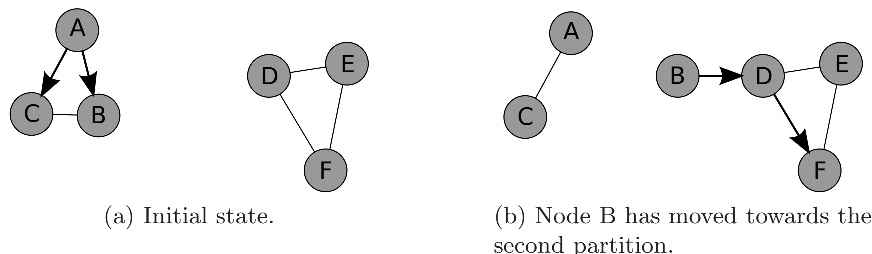
Figure 2.3: Message forwarding in LSR versus OLSR. Link-state information from the white middle node is broadcast. Light gray nodes forward messages, black nodes do not.

topology changes. The optimization introduced by OLSR imposes distribution of link-state information to only a subset of a node's neighbors, called the node's *multi-point relay* (MPR) set. The MPR set for a node is the minimal subset of the node's neighbors which must re-broadcast a message for it to reach all of the node's two-hop neighbors.

A node's link-state information is broadcast to all its neighbors, but only those in the node's MPR set re-broadcasts the information, minimizing the number of messages sent during route discovery and maintenance. Also, only the link-states of the neighbors in a node's MPR set is advertised, as opposed to the entire set of neighbors. This is sufficient, as each of the node's two-hop neighbors is a one-hop neighbor of some node in the MPR set [13].

The effect of this optimization is shown in Figure 2.3. Figure 2.3a shows the message forwarding in LSR, where every node re-broadcasts all received link-state messages, while Figure 2.3b shows the message forwarding in OLSR, where only members of a node's MPR set re-broadcasts a link-state message.

Ad-Hoc On Demand Distance Vector Routing *Ad-hoc On-demand Distance Vector* (AODV) is, as opposed to OLSR, a *uniform*, *destination-based*, *reactive* routing protocol [13]. Whenever a node needs a route to a certain destination, it broadcasts a *route request* (RREQ). The RREQ is re-broadcast at every node until it reaches the correct destination. Then, a *route reply* (RREP) is sent back towards the source. On the way back, a forward destination vector is created at each intermediate node. Data destined for the located node then follows the path stated in the forward destination vector.



(a) Initial state.

(b) Node B has moved towards the second partition.

Figure 2.4: Epidemic routing in a MANET with two partitions and one node moving from the sending partition to the receiving partition. The messages sent from A towards F follows the arrows.

Routing in Sparse MANETs

If the application can tolerate it, one might use *delay tolerant* techniques to support routing. These techniques are designed to handle partitioning and the lack of a path between two nodes by trying to find a *time-space* path between the sender and the receiver. A time-space path is a path which does not exist at one single point in time but rather over a time interval.

These solutions follow the *store — carry — forward* principle, which means that when a node receives a message that cannot immediately be delivered, it is stored, carried around with the node and then, whenever possible, forwarded, either to another carrier or to the destination. Messages may be stored for an arbitrary amount of time, but by allowing the time between sending and receiving data to be arbitrary large, the probability of correct delivery gets very close to one.

Epidemic Routing *Epidemic routing* is a reactive approach, in which messages are distributed to *carriers*. The carriers are responsible for distributing the message throughout the network [41]. The approach relies on the carriers to, at some point in time, get within range of other partitions so that the messages they carry may be forwarded to nodes in these partitions. The approach is illustrated in Figure 2.4, showing a network consisting of two partitions. In Figure 2.4a, node A, located in the first partition, is ready to send a message to node F, located in the second partition. Node A then forwards its message to the nodes within range, B and C, which then carry this message with them whereever they go. In Figure 2.4b, node B has moved towards the second partition, and may thus exchange carried messages with node D. Node D may now forward the message to its neighbor, node F.

Message Ferrying *Message Ferrying* is a proactive routing protocol. In message ferrying, special nodes, called *message ferries* or just *ferries*, are used to deliver messages. Ferries move along a non-random path, picking up and delivering messages to the regular nodes it passes [44]. The message

ferrying scheme may be either *node initiated* or *ferry initiated*. In a node initiated scheme, the ferry path is known to all nodes. Whenever a node wants to send something, it moves towards the ferry and delivers its messages to the ferry when the ferry is within the node's range.

In the ferry initiated scheme, the ferry moves according to the nodes' needs. Long range radio is used by nodes to inform the ferry that they are ready to transmit, instructing the ferry to move towards these nodes.

2.1.4 Applications

The term *ad-hoc* comes from Latin and means *for this purpose*. A MANET is a network which may be set up anywhere in a short time, thus being able to satisfy a demand at short notice. A MANET is set up when no infrastructure is present, thus a MANET is the only possibility for communication, or whenever the use of present infrastructure is unwanted, for example for security reasons or simply because the existing infrastructure is too expensive to use.

As there is no centralized control within a MANET, there is no single point of failure. This, together with easy setup and self-organizing nodes, makes MANETs robust to errors and topology changes in terms of entering and leaving nodes.

MANETs may be suitable in a wide variety of settings. Some typical applications include:

- *Rescue operations*, as illustrated in the motivation for this thesis in Section 1.2.
- *Vehicular environments:* MANETs may be used in vehicular environments, for example to share information between vehicles. If an accident or some other obstacle has occurred, passing cars may transmit information about this to other vehicles on their way to this area, making them aware of the situation.
- *Wildlife monitoring:* When monitoring animals in their natural habitat, the monitoring should not interfere with the animals' natural behavior. Thus, animals should be able to move as usual and should not be bothered by the sensing device in any way, as this might affect their behavior. To achieve this, small sensing devices may be used. These run on battery power, and a sensor on one animal may communicate both with sensors on other animals as well as with base stations or nearby researchers. An example of such a use is *ZebraNet* [19], which tracks animal data such as animal location in an area of hundreds or thousands of square kilometers in Kenya.
- *Military operations:* Military operations often take place in areas where infrastructure is non-existing or not suitable to use because of security

issues. The use of a MANET is such scenarios would enable military personnel to communicate, track friendly and enemy forces, and pinpoint hazards like minefields. This field is the originator of the basic ad-hoc network techniques [38]. MANETs have been used for this purpose by the American forces in Iraq [32].

- *Home networking and personal entertainment:* MANETs may be set up in personal homes, classrooms and public areas to enable for example file sharing and gaming.

2.2 Autonomic Networking

2.2.1 Autonomic Networks

As stated in the introduction, what we want is for networks to have the ability to organize themselves, and to adapt to the changes in themselves and their environment. This kind of networks may also be called *Autonomic Networks*. Schmid et al. [34] propose the following definition of autonomic systems:

> "An *Autonomic System* is a system that operates and serves its purpose by managing its own self without external intervention even in case of environmental changes."

In other words, an autonomic network should be able to *detect* changes and to *react* upon them.

2.2.2 Self-*: Properties of an Autonomic Network

An autonomic network should be self-organizing. This means that the nodes in the network should organize themselves, and cooperate to form a community through dynamic role assignment and joint decision making.

Exactly what properties are needed in an autonomic network is discussed in the literature. Schmid et al. state in [34] that all autonomic systems need to exhibit the following minimal set of properties to be able to actually function autonomically:

- Automatic: The system, here the network nodes, needs to self-control its internal functions and operations, including bootstrapping, without any manual intervention.

- Adaptive: The system needs to be able to change its operation to be able to cope with temporal and spatial changes in its context.

- Aware: At last, the system needs to be able to monitor its context as well as its internal state.

Serugendo [35], as well as Kephart and Chess [20] identify four fundamental principles that characterize an autonomic network. These principles are often called the *self-** properties, and are stated as follows:

- Self-configuration:

  > "Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly." [20]

  This is the first step of self-management within an autonomous network. It includes adjusting to new and updated components, as well as leaving components. For example, the nodes in a network need to adapt to new nodes connecting to the network, as well as nodes leaving the network.

- Self-optimization:

  > "Components and systems continually seek opportunities to improve their own performance and efficiency." [20]

  It is important that all nodes *cooperate* to keep the network in a state that is feasible not only for one node, but for the network as a whole. Self-optimization in a network needs to be done at both node and network level. On node level, this means for the node to adapt to the current conditions both in the node's environment, meaning the rest of the network, as well as in the node itself. On network level, this includes global optimization through joint decision making.

  For example, a node needs to adjust its own resource usage, both according to its own needs and to the needs of the other nodes in the network. If not, we risk having one node using all the available bandwidth, leaving all other nodes starving.

- Self-protection:

  > "System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system wide failures." [20]

  As networks and computer systems in general are getting more and more complex, the probability of some problem occurring also increases. Such problems may be both malicious attacks and normal software and/or hardware failures. To prevent the system or network from complete failure, it is important for the nodes in the network to be able to protect themselves against these threats.

- Self-healing:

> "System automatically detects, diagnoses and repairs localized software and hardware problems." [20]

Self-healing needs, as self-optimization, to be performed at both node level as well as network level. On node level, each node can recover from failure by re-configuring itself, e.g. by replacing a failed component with an equivalent, well-functioning component. On network level, recovery may be accomplished by re-organizing the network, e.g. by replacing failing nodes with other nodes capable of conducting the same tasks.

For the network nodes to be able to fulfill these four self-* properties, they obviously need quite some knowledge, both about their own state as well as the network state. Thus, each node also needs a fifth property, namely *self-monitoring* — it needs to be able to monitor both itself and the rest of the network.

- Self-monitoring: Nodes need to monitor themselves to make sure they fulfill their objectives. Exactly what to monitor depends on the needs of the node and of the network. Also, the nodes somehow need to monitor their environment to be able to cooperate with the other nodes in a best possible way. In other words, they need to be *context-aware*. In [7], Dey gives the following definition of context:

  > "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

  In [39], metadata, more specifically *profiles* and *policies*, are viewed as an emerging approach to support context-awareness. Profiles represent data concerning users, devices, system components and the surrounding environment. "Data" may include information like user preferences, device capabilities (available disk space, available memory, installed software) and network conditions. Policies, on the other hand, express system behavior, in terms of the actions subjects can or must operate upon resources. In [37], Sloman distinguishes between two kinds of policies: *Authorization* policies and *Obligation* policies. The former describes which actions are *allowed* and which are not, while the latter describes which actions are *mandatory* and which are not.

  Monitoring is also important when trying to detect problems, or when recovering from already occurred problems.

We see that the properties stated in [34] cover mostly the same as those proposed by [35, 20]. Schmid's *automatic* property corresponds well to the

self-configuration property, while the *adaptive* property covers both *self-optimization*, *self-protection* and *self-healing*. At last, the *aware* property and the *self-monitoring* property are equivalent.

2.2.3 Autonomy in MANETs

We have already stated that a MANET is a collection of, amongst other things, *autonomous* nodes. For a MANET to be able to fulfill its purposes, nodes need to be autonomous. The network should be easy to set up, requiring self-organizing node properties. The MANET also needs to be able to maintain itself, requiring self-configuration, self-optimization and self-protection, along with self-healing if anything should go wrong.

Chapter 3

A Bug's Life

Calvin: "That's the problem with nature. Something's always stinging you or oozing mucus on you. Let's go watch TV."

— *Bill Watterson*

A lot of self-organizing systems exist in nature. These function without any external or central control, and thus resemble the autonomous systems we are striving to develop for MANETs.

A lot of complicated computing problems concern *optimization* - how to choose the best alternative from a set of possible solutions to a given problem. Some practical problems include time tables and scheduling, telecommunication network design and shape optimization [3]. In computer networking, the probably most well known problem is *routing* - to find paths between nodes within a network. We do not want any path, we want the *best* path.

Many such problems have been simplified in order to obtain scientific test cases. An example of such a simplified test case is *the traveling salesman* problem.

In this chapter, we dive into the world of biology and take a look at ants and how they manage to co-operate and organize their flock without any direct communication. Such animal behavior may be used as inspiration when designing computer systems such as our resource localization system. To understand how, however, we need to know how the animals function themselves. In the next chapter, we take a deeper look at the computer science aspect, and will see some example solutions to some well known optimization problems.

3.1 Communication

Before we start our discussion of ants and stigmergy, a few terms and definitions need to be clarified. Central in this topic is *communication*, or

interaction between individuals.

> ”Communication: A process by which information is exchanged between individuals through a common system of symbols, signs, or behavior <the function of pheromones in insect communication>; also : exchange of information”
> (Merriam-Webster OnLine Dictionary)

In [14, p. 1], *interaction* is defined as ”the ongoing two-way or multiway exchange of data among computational entities, such that the output of one entity may causally influence the outputs of another”.

We may further divide into *direct* and *indirect* interaction. Direct interaction happens via *messages*, or *message passing*, where the recipient's identification is included in the message. However, as we shall see in this chapter, some individuals rather communicate via *indirect interaction*, which is ”interaction via *persistent*, observable state changes” [14, p. 1]. The recipients of this kind of ”messages” are any individuals observing these state changes. This kind of communication may exhibit a lot of characteristics not present in message passing [14]:

- *Late binding of recipient:* The identity of the recipient is not necessarily known at the time of the state change.
- *Anonymity:* The identity of the recipient is not necessarily known to the originator of the state change.
- *Time decoupling (asynchrony):* State changes may be persistent in the environment, and there may thus be a delay between the state change and state change observation.
- *Space decoupling:* The state change originator and observer need not be co-located - They only need to visit the same spot at some point in time.
- *Non-intentionality:* The state change originator does not necessarily have the intention to communicate.
- *Analog nature:* The medium of the indirect interaction may be the real world.

When explaining the design of our ant solution in Chapter 5, we will revisit these characteristics and look at how they relate to our approach.

3.2 Stigmergy and Ant Colony Optimization

As stated in section 1.3, the term stigmergy was introduced by Grassé in 1959 and defined as

> "Stimulation of workers by the performance they have achieved."
> Dorigo et al. [9]

Grassé studied the social behavior of termites, and found that these cooperate on performing different tasks without any direct interaction. Indirect communication may be performed through for example environmental changes, more specifically by depositing *pheromones*.

3.2.1 Pheromones

> "A chemical substance that is usually produced by an animal and serves especially as a stimulus to other individuals of the same species for one or more behavioral responses."
> (Merriam-Webster OnLine Dictionary)

The term *semiochemicals*, which is derived from the Greek word *semeion - sign*, is used for chemicals involved in animal communication. *Pheromones* are a subclass of semiochemicals used in intraspecific communication [42].

The term "pheromone" is derived from the Greek words *pherein*, which means *"to bear"* and *hormōn*, which means to *excite* or *stimulate*. The term was introduced in 1959 by Peter Karlson and Martin Lüscher, who were working on identifying the chemicals that maintain the caste system of termites [27].

According to Wyatt [42], pheromones were originally defined by Karlson and Lüscher as "substances secreted to the outside by an individual and received by a second individual of the same species in which they release a specific reaction, for instance a definite behavior [releaser pheromone] or developmental process [primer pheromone]".

As stated in the above definition, pheromones may be used by animals to attract other individuals of the same species. For example, animals like cats and dogs leave *territorial marking pheromones*; The cat by rubbing its cheek on a human leg, and the dog by urinating. *Alarm pheromones* are left by aphids whenever an individual gets crushed, making other nearby aphids flee. Some species leave *kin-recognition pheromones* to help each individual recognize which other individuals are family and which are not. Other species, for example the *Aphaenogaster rudis* ants, leave *recruitment pheromones* to lead nest mates to a food source. (All examples fetched from [6]).

3.2.2 Ants and Their Pheromones

For us humans, sight and hearing are the most important senses. For many ant species, however, the sight sense is only rudimentary developed, and some ant species are even completely blind [10]. Thus, ants can not necessarily rely on this sense when communicating with each other. Instead, they

(a) Lasius Niger, from `http://harrierpestprevention.info/about`

(b) Iridomyrmex humilis, from `http://www.terro.com/guide-ants.php`

Figure 3.1: Two kinds of ants leaving pheromone trails

communicate indirectly with pheromones. Ants make use of different kinds of pheromones, but particularly important is the *trail pheromone* used by some ant species, such as *Lasius niger* (black garden ant), shown in Figure 3.1a and *Iridomyrmex humilis* (Argentine ant), shown in Figure 3.1b. These ants use pheromones to leave trails on the ground, which in turn may be used for example to record the path to a food source. Other ants may later smell these trails and walk the same path, and are thus lead to the food source.

What makes the pheromone trails particularly useful, is that ants tend to probabilistically choose the paths with the highest pheromone concentrations. Initially, ants will walk a random path, as no pheromone traces are present. Ants walking leave pheromones along the path they are walking. The pheromone concentration along one path will decrease with time because of diffusion. The speed of this diffusion is such that over time, shorter paths will get a higher pheromone concentration, as these paths take a shorter time for an ant to walk. The higher concentration makes more ants choose this particular path, again leading to an even higher concentration of pheromones on this path. After a while, the same, shortest path will be used by most ants. There will, however, always be a chance for another path to be chosen.

Pheromones *evaporate* over time. Pheromone evaporation enables a form of forgetting in the sense that too rapid convergence towards a suboptimal region is avoided. Also, if the food situation should change, old and outdated information will disappear with time.

However, studies with real ants have shown that in some cases, ants are unable to converge to the shortest path [10]. In the study, the ants were offered only one path from their nest to a food source. Thus, all ants walked this path to get to the food. After 30 minutes, a shorter branch was added, as shown in Figure 3.2. One would think that now, the ants would move from the longer path to the new and shorter path. However, only a small number of ants chose the shorter branch, and the colony was never able to converge to using the new path. This happens due to the characteristics of

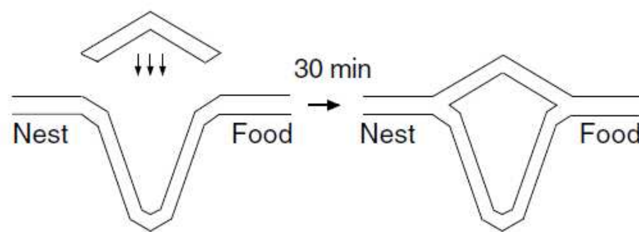


Figure 3.2: Experiment setup where ants are unable to converge to the shortest path. Figure copied from [10, p. 5].

pheromone evaporation: Evaporation on the longer branch happens too slow for high pheromone concentration to decrease, and the longer branch is still reinforced after the shorter path was introduced.

At least one ant specie, namely the *Monomorium pharaonis* (pharaoh ant), also leaves *repellent pheromones* when they find that a path does *not* lead to a food source. This kind of pheromone will work as a "no entry" signal, marking the unrewarding branch with a signal which greatly increases the probability of other ants selecting a different branch or making a U-turn [33].

3.2.3 Ant Colony Optimization

From the above discussion, we see that the ants do not only find a path from their nest to the food source, they actually tend to find the *shortest path* to the food source. This fact has inspired computer scientists to develop algorithms to solve optimization problems. The first such attempts were done in the early 1990s, and one of the outcomes of this research is *ant colony optimization* (ACO). ACO algorithms are now "the most successful and widely recognized algorithmic technique based on ant behaviors" [10, p. ix].

An example of an ACO application area is network management, such as routing and load balancing. How ACO algorithms work and may be used to solve such problems will be further discussed in Chapter 4.

Chapter 4

ACO Approaches to Some Traditional Problems

"The system of nature, of which man is a part, tends to be self-balancing, self-adjusting, self-cleansing. Not so with technology."

— *E.F. Schumacher*

This chapter will give some samples on how ACO algorithms may be used to solve some traditional computing problems, namely the *traveling salesman problem* and the *routing problem*, both in traditional, wired networks as well as in MANETs.

Note that the contents in this chapter are partly based on the book "Ant Colony Optimization" by Dorigo and Stützle [10], and the contents in Section 4.3.2 are also partly based on the article "Anthocnet: an ant-based hybrid routing algorithm for mobile ad hoc networks" by Di Caro, Ducatelle, and Gambardella [8].

4.1 The Ant Colony Optimization Metaheuristic

A *metaheuristic* is "a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems" [10, p. 25]. In [10], Dorigo and Stützle define the ant colony optimization (ACO) metaheuristic, inspired by the behavior of real ants. In this metaheuristic, artificial ants cooperate on finding good solutions to discrete optimization problems. In the following, we will provide a short summary of the ACO metaheuristic. Please note that, for simplicity, a lot of details and formalities have been left out.

In ACO, an artificial ant is "a stochastic procedure that incrementally builds a solution by adding opportunely defined solution components to a

partial solution under construction" [10, p. 34]. Solutions are built by the artificial ants, which are moving on the *construction graph* $G_C = (C, L)$, where the set of arcs L fully connects the components C. Each component $c_i \in C$ and connections $l_{ij} \in L$ can have associated a *pheromone trail* τ, which may be associated either with components, then denoted τ_i, or connections, then denoted τ_{ij}, and a *heuristic value* η (η_i and η_{ij}). The pheromone values are *long term memory* about the entire search process, whereas the heuristic values represent a priori information about the problem instance or run-time information provided by a source different from the ants.

Each artificial ant k exploits the construction graph to search for optimal solutions. The ant has a memory M^k where it stores information about the path it has followed. This memory is used both to build solutions, to compute heuristic values, to evaluate solutions and to retrace the path to find the way back.

Each ant has a *start state* and a set of *termination conditions*. If at least one termination condition is satisfied, the ant stops. If no termination condition is satisfied, the ant moves to a node in its neighborhood. Which neighbor to move to is decided by applying a probabilistic decision rule, which is a function of the locally available pheromone trails and heuristic values, the ant's private memory and the problem constraints. When a component is added to the ant's current state, the ant may update the pheromone trail τ associated with either the component or the corresponding connection. When a solution has been built, the ant retraces the traveled path and updates the pheromone trails of the used components.

As a summary, an ACO algorithm may be imagined as consisting of three procedures:

- **ConstructAntsSolutions:** Manages the ant colony. Each ant visits adjacent states of the considered problem by moving through neighbor nodes. Neighbor selection is done by stochastically selecting a next hop according to pheromone trails and heuristic information associated with each arc or the node as a whole. Each ant evaluates its solution, either during building or after a complete solution has been built.

- **UpdatePheromones:** Modifies pheromone trails. Pheromone concentration increases when pheromones are "deposited", and decreases with time due to pheromone evaporation.

- **DaemonActions:** Used to implement central actions which cannot be performed by single ants, such as activation of a local optimization procedure or collection of global information.

In Figure 4.1, we reproduce a short, general pseudo-code found in [10, p. 38]. This pseudo-code does not give any information on how the three procedures should be executed in relation to each other. This issue is completely up to the system designer.

```
procedure ACOMetaheuristic
  ScheduleActivities
    ConstructAntsSolutions
    UpdatePheromones
    DaemonActions          % optional
  endScheduleActivities
endProcedure
```

Figure 4.1: Pseudo-code for the ACO metaheuristic.

In the metaheuristic, as in real life, the ants communicate indirectly via the pheromone trail values. Thus, this may be viewed as "a distributed learning process in which the single agents, the ants, are not adaptive themselves but, on the contrary, adaptively modify the way the problem is represented and perceived by other ants" [10, p. 37].

4.2 The Traveling Salesman Problem

The *Traveling Salesman problem* (TSP) is an $\mathcal{NP}$-hard problem in combinatorial optimization. In this problem, a salesman has a set of towns he is going to visit. He starts from his home town, and wants to find the shortest possible path such that each city is visited once and only once.

This problem may be represented by a complete weighted graph $G = (N, A)$ where N is the set of nodes or cities to be visited and A is the set of arcs connecting the nodes. Each arc (i, j) is assigned a weight d_{ij}, representing the distance between the two cities i and j. The TSP is then the problem of finding a minimum length Hamiltonian circuit of the graph, where the Hamiltonian circuit is a closed walk visiting each node n of G exactly once. An optimal solution to this problem is thus a permutation π of the node indices $\{1, 2, \ldots, n\}$ such that the length $f(\pi)$ is minimal, where $f(\pi)$ is given by

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \tag{4.1}$$

[10, p. 66]. Note that the absolute position of a city in a tour is not important, only the relative order is, making n permutations map to the same solution.

4.2.1 Solving the TSP with ACO Algorithms

When solving the TSP with ACO, we need to map the characteristics of the TSP to the ACO metaheuristic. The *construction graph* is identical to the problem graph, as the number of components correspond to the set of nodes.

The connections correspond to the set of arcs, and the connection weights correspond to the distance d_{ij} between the two nodes i and j.

The TSP involves only one *constraint*, namely that all cities must be visited exactly once. The feasible neighborhood for an ant choosing its next hop thus comprises all cities that are still unvisited.

Pheromone trails, denoted τ_{ij}, in the TSP correspond to the desirability of visiting city i right after city j. The *heuristic information* η_{ij} is usually inversely proportional to the distance between city i and city j, for example $\frac{1}{d_{ij}}$.

A *solution* is built by placing each ant on a randomly chosen start city. At each step, each ant iteratively adds one still unvisited city to its partial tour. As soon as all cities have been chosen, the solution construction is terminated.

4.2.2 AntSystem

The TSP has played a central role in ACO, as this was the application problem chosen for the first proposed ACO algorithm, namely *AntSystem* (AS). The TSP has also been used as test system for most ACO algorithms developed later [10]. AS has been found to be inferior to state-of-the-art algorithms for the TSP, but has still worked as an inspiration for later, more efficient systems extending AS, such as $\mathcal{MAX} - \mathcal{MIN}$ AS, elitist AS and rank-based AS. However, in this thesis, we choose to focus on AS rather than the extensions, as this gives the clearest picture of the ACO concepts. The main difference between AS and the mentioned extensions lies in the way the pheromone update is performed.

Initially, there were three different versions of AS: *ant-density*, *ant-quantity* and *ant-cycle*. In the two former approaches, pheromone values were updated directly after a move between two nodes, whereas in the latter version, pheromone values were updated only after all ants had constructed the tours, making the amount of deposited pheromone reflect the quality of the tour. However, AS is now synonym with the latter approach, as this approach turned out to outperform the other two.

Tour Construction

Initially, m ants are placed on randomly chosen cities. At each construction step, ant k chooses its next hop by applying a probabilistic action choice rule, called *random proportional* rule. The probability p for ant k, currently located at city i to choose as next hop city j is

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \tag{4.2}$$

where η_{ij} is a heuristic value that is available a priori, α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information, and $\mathcal{N}_i^k$ is the feasible neighborhood of ant k being in city i, meaning the set of cities not yet visited by k. For details on what values to use for these parameters, see [10, p. 71].

A memory, $\mathcal{M}^k$, is maintained by each ant k. This memory contains which cities have been visited so far, in the order they were visited. The memory is used to define the feasible neighborhood $\mathcal{N}_i^k$, to find the length of a tour and to retrace the path to deposit pheromone after construction.

Pheromone Values

An AS heuristic to pheromone trail initialization is to set them to a value slightly higher than the expected amount of pheromone deposited by the ants in one iteration. A rough estimate to this value can be estimated by setting, $\forall(i,j), \tau_{ij} = \tau_0 = \frac{m}{C^{nn}}$, where m is the number of ants and C^{nn} is the length of a tour generated by the nearest-neighbor heuristic, which always chooses the nearest (unused) neighbor as next-hop.

After all tours have been constructed, pheromone values are updated. Pheromone evaporation is executed first by lowering the pheromone values on *all* arcs by a constant factor:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}, \quad \forall(i,j) \in L \tag{4.3}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate and L is the set of arcs in the construction graph. After evaporation, all ants deposit pheromones on the arcs they have crossed:

$$\tau_{ij} \leftarrow \tau_{ij} + \textstyle\sum_{k=1}^{m} \Delta\tau_{ij}^k, \quad \forall(i,j) \in L \tag{4.4}$$

where $\Delta\tau_{ij}^k$ is the amount of pheromone ant k deposits on the arcs it has visited. It is defined as follows:

$$\Delta\tau_{ij}^k = \begin{cases} 1/C^k, & \text{if arc } (i,j) \text{ belongs to } T^k, \\ 0, & \text{otherwise;} \end{cases} \tag{4.5}$$

where C^k is the length of the tour T^k, computed as the sum of the lengths of the arcs belonging to T^k. Equation (4.5) gives higher pheromone deposits the better the tour is.

4.3 The Routing Problem

The *routing problem* refers to "the distributed activity of building and using *routing tables*" [10, p. 224]. Routing tables are present in each node in the network, and contains information used by the *routing algorithm* to make

local forwarding decisions, more specifically which outgoing interface to forward incoming packets on. The characteristics of networks make the routing problem harder to solve; nodes move, disappear and appear, making links tentative. Thus, this problem is quite different from the TSP above. The routing problem is also, however, an optimization problem, as we are always interested in the *most optimal* path from A to B. What an *optimal* path is, however, may be discussed. This discussion is, however, outside the scope of this thesis.

4.3.1 AntNet — Routing in Traditional Networks

An example of use of stigmergy in traditional networks is *AntNet* [4]. AntNet is an approach to the adaptive learning of routing in communications networks. The approach is based on artificial *agents*, simulating the ants. The agents are used to conduct repeated simulations, where information collected from past simulations are used to improve the results.

Problem Description

AntNet is targeted at irregular topology packet-switched data networks with an IP-like network layer and a very simple transport layer, and focuses on wide area networks (WANs).

The network instances considered by AntNet can be mapped to directed weighted graphs with n forwarding nodes. All links between nodes may be viewed as bit pipes characterized by a bandwidth, measured in bits per seconds, and a transmission delay, measured in seconds. Packets are divided between data packets, which are stored in low-priority queues within each node, and routing packets, which are served in high-priority queues.

The results from [4] state that AntNet shows superior performance and robustness for most of the experiments conducted. In [10] (published in 2004), Dorigo and Stützle state that AntNet is "the sole algorithm to have reached, at least at the experimental/simulation level at which it was tested, state-of-the-art performance" [10, p. 223].

Algorithm

The algorithm used in AntNet is highly representative for stigmergy-inspired algorithms, and may be summarized as follows:

1. At regular intervals, from each network node, artificial *forward ants*, are sent towards a randomly selected destination node. These agents are supposed to find the most feasible path from the source to the destination. In each node along the path, the forward ants share queues with the data traffic, making the agents experience realistic traffic loads.

2. Artificial ants act concurrently and independently, and communicate indirectly as explained above.

3. Each forward ant searches for a minimum cost path between the source and destination nodes.

4. Each forward ant moves towards the destination node, hop-by-hop. At each intermediate node, the next hop is chosen by applying a greedy, stochastic policy, making use of pheromone values recorded in the current node, the ant's memory and any node-local problem-dependent heuristic information.

5. All forward ants keep record of their traveled path, the experienced traffic conditions and time length.

6. When the destination node is reached, the forward ant creates a new agent, a *backward ant*, transfers to it all of its memory, and dies. The backward ant travels back towards the source along the path recorded by the forward ant.

7. Along the way back, the backward node updates pheromone values in the nodes along its path with the new information learned by the forward ant.

Solution Construction

At regular intervals Δt a forward ant $F_{s \to d}$ is launched from each node s toward a destination node d to discover a feasible path between the two nodes, as well as to investigate the load status along the path. Forward ants are treated as data packets, thus sharing with them a low-priority queue, which gives them the same experience of network traffic as later data packets would. Destinations are locally selected according to the data traffic patterns generated by the local workload: if f_{sd} is a measure (in bits or the number of packets) of the data flow workload, then the probability of creating at node s a forward ant with node d as destination is

$$p_{sd} = \frac{f_{sd}}{\sum_{i=n}^{n} f_{si}} \tag{4.6}$$

This distribution makes ant exploration adapt to the variance in data traffic, thus looking more often for new paths to often used destinations.

When choosing the next hop, forward ants choose between all neighbors that have previously not been visited, or all neighbors if all neighbors have been visited. The probability of choosing node j as next hop is p_{ijd}, computed as the normalized sum of the pheromone value τ_{ijd} with a heuristic

value η_{ij} taking into account the length of the j-th link queue of the current node i:

$$p_{ijd} = \frac{\tau_{ijd} + \alpha\eta_{ij}}{1 + \alpha(|\mathcal{N}_i| - 1)}. \tag{4.7}$$

The heuristic value η_{ij} is a $[0, 1]$ normalized value function of the length q_{ij} (in bits waiting to be sent) on the queue on the link connecting the node i with its neighbor j:

$$\eta_{ij} = 1 - \frac{q_{ij}}{\sum_{l=1}^{|\mathcal{N}_i|} q_{il}} \tag{4.8}$$

The value of α weighs the importance of the heuristic value with respect to the stored pheromone values.

If cycles are detected, meaning that the forward ant has returned to an already visited node, the cycle's nodes are removed from the ant's memory. Forward ants have a *max_life* parameter telling how many hops a forward ant may travel. If the destination is not reached within this number of hops, the ant dies.

Pheromone Values

In this section we give a brief overview of pheromone value updating in AntNet. Please note that for simplicity, a lot of details have been omitted.

Pheromone values, as well as traffic models, are updated by backward ants. Pheromone values are updated by incrementing the pheromone $\tau_{ifd'}$ (the pheromone suggesting to choose neighbor f when the destination is d') and decrementing the other pheromones $\tau_{ijd'}$, $j \in \mathcal{N}_i$, $j \neq f$. Pheromone updates depend on a measure of the goodness associated with the trip time $T_{i \to d'}$ experienced by the forward ant. In this thesis, we have chosen not to dig into the details of goodness computation. For more information on this, see [10, ch. 6].

The reinforcement $r \equiv r(T, \mathcal{M}_i), 0 < r \leq 1$, where $\mathcal{M}_i$ is the local traffic model, is used to update the pheromones. It is used by the backward ant $B_{d \to s}$ moving from node f to node i to increase the pheromone values $\tau_{ifd'}$:

$$\tau_{ifd'} \leftarrow \tau_{ifd'} + r \cdot (1 - \tau_{ifd'}). \tag{4.9}$$

Small pheromone values are thus increased proportionally more than large pheromone values, favoring a quick exploitation of new, and good, discovered paths.

Pheromones $\tau_{ijd'}$ for destination d' of the other neighboring nodes j, $j \in \mathcal{N}_i$, $j \neq f$, evaporate implicitly by normalization: Their values are reduced so that the sum of pheromones on links exiting from node i will remain 1:

$$\tau_{ijd'} \leftarrow \tau_{ijd'} - r \cdot \tau_{ijd'}, j \in \mathcal{N}_i, j \neq f. \tag{4.10}$$

4.3.2 AntHocNet — Routing in Mobile Ad-Hoc Networks

As stated in Chapter 2, nodes in a MANET should be capable of working as routers as well as ”normal” network nodes, as there is no fixed infrastructure in such a network. A large number of algorithms for routing in MANETs exist. Common for the existing algorithms is, however, that they require large amounts of overhead to perform the routing procedure [23]. In MANETs, resources tend to be scarce, and the amount of overhead is thus wanted to be as small as possible.

In [23], Mehfuz and Doja argue that ACO algorithms are well suited also for MANET routing algorithms. They identify four properties of this kind of algorithms that illustrate why this is the case:

- *Dynamic topology:* This is one of the major properties of MANETs, and thus needs to be supported by MANET routing algorithms. As ant colony optimization algorithms are based on individual agents, or ants, this allows for a high adaptation to the rapidly changing network topology.

- *Local work:* Instead of flooding topology information throughout the network, these algorithms are based only on local information, giving a great reduction in the amount of control overhead.

- *Link quality:* The ants leave traces on the nodes they visit, and these traces are later used to determine the next hop by other ants. These traces may contain information about link quality, so that routing decisions may be taken based also on link quality, not only hop count, which is usual in MANET routing algorithms.

- *Support for multipath:* All nodes keep ”ant trace” information for all their neighbors.

AntHocNet [8] is an ACO inspired algorithm for routing in MANETs. In AntHocNet, ants follow and update pheromone tables in a *stigmergic learning* process. Data packets are routed stochastically according to the learned tables [5]. However, routing in traditional, wired networks is mostly done proactively. Because of MANET characteristics, AntHocNet exploits a *hybrid approach*: The approach is *reactive* in the sense that route information is only gathered when some node actually needs it, while it is also *proactive* because nodes try to maintain and, if possible, improve routes for the duration of the communication between the source and destination. There are thus two phases: *path setup* and *path maintenance and improvement*. The latter phase is supported by a process called *pheromone diffusion*, where obtained routing information is spread between the nodes in the MANET in an information bootstrapping process.

Solution Construction

Like AntNet, AntHocNet uses *forward* and *backward* ants. Whenever a node s needs a path to a destination d and node s has no available routing information for d, it *broadcasts* a *reactive forward ant.* At each intermediate node, this ant is either *unicast* or *broadcast*, according to whether or not the current node has any pheromone information for d. If no pheromone information is available, the forward ant is broadcast. If a node receives multiple broadcast copies of one forward ant, only the first copy is handled. If pheromone information is present, the next hop n is chosen with the probability P_{nd}. This probability depends on the relative goodness, which is a combined measure of path end-to-end delay and number of hops, of n as a next hop, expressed in the pheromone variable $\mathcal{T}^i_{nd}$:

$$P_{nd} = \frac{(\mathcal{T}^i_{nd})^\beta}{\sum_{j\in\mathcal{N}^i_d}(\mathcal{T}^i_{jd})^\beta} \quad \beta \geq 1, \tag{4.11}$$

where $\mathcal{N}^i_d$ is the set of neighbors of i over which a path to d is known, and β is a parameter which controls the exploratory behavior of the ants.

Proactive Path Maintenance After a path has been set up, source nodes send out *proactive forward ants* to update the information about currently used paths and to try to find new and better paths. These ants follow pheromone and update routing tables in the same way as reactive forward ants. To avoid excessive bandwidth consumption through random walks and broadcasts, *pheromone diffusion* is introduced to allow spreading of pheromone information throughout the network. This is done by periodically and asynchronously broadcasting short messages by the nodes to all their neighbors. These short messages contain a list of destinations the sending node has routing information for, including for each of these destinations d the best pheromone value $\mathcal{T}^d_{m*d}, m* \in \mathcal{N}^n_d$, which n has available for d. All receiving nodes may use this information to update their own routing and pheromone tables, as well as for path exploration. For more details on path maintenance and exploration, see [5].

Pheromone Values

Each ant maintains a memory of the nodes it has visited. This memory is used by backward ants to backwards retrace the path followed by the forward ant. At each node along this path, the backward ant reads a locally maintained estimate $\hat{T}^i_{i+1}$ of the time it takes to reach the neighbor $i+1$, the node the backward ant arrived from. The time $\hat{T}^i_d$ it would take a data packet to reach d from i over the path from the ants memory is calculated incrementally as the sum of the local estimates $\hat{T}^j_{j+1}$ gathered by the ant between i and d. This time estimate $\hat{T}^i_d$ is combined with the number of

hops h between i and d over the recorded path to calculate the pheromone value τ_d^i by the following formula:

$$\tau_d^i = \left(\frac{\hat{T}_d^i + hT_{hop}}{2} \right)^{-1}, \tag{4.12}$$

where T_{hop} is a fixed value representing the time to take one hop in unloaded conditions. The estimated *goodness* of going from node i over neighbor n to reach destination d, $\mathcal{T}_{nd}^i$, is then updated as follows:

$$\mathcal{T}_{nd}^i = \gamma \mathcal{T}_{nd}^i + (1 - \gamma)\tau_d^i, \quad \gamma \in [0, 1]. \tag{4.13}$$

Chapter 5

Design

> *"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."*
>
> — *Douglas Adams*

In this chapter, we will derive a design for an ant-inspired algorithm for locating a resource in a MANET. The first sections state the goal, the assumptions made and the requirements for our solution, before we start the description of the issues and design choices made during our work. The final section provides a short pseudo code summary of the design.

5.1 Goal

As stated in the problem description in Section 1.3, what we want to achieve is an algorithm for locating a resource within a MANET. Certain nodes may contain resources useful to other nodes. In some situations, there is no existing framework for spreading such information, or there is no way to predict what kind of information needs to be spread when designing applications. In such situations, interested nodes need a way to find these resources themselves whenever they need them. As nodes should function autonomously, the resource localization should take place without any outside intervention.

5.2 Assumptions

We assume random placement of both resource-requesting nodes as well as resource-holding nodes within the MANET. Thus, our nodes have no initial knowledge about the resource location. Also, we assume that a node's resources are independent of the node's physical location, thus, the fact that a node may be mobile does not inflict with its resources.

5.3 Requirements

There are some absolute requirements that our solution needs to fulfill. These reflect the purpose of the system, and the environment in which the system is designed to run:

- If a requested resource is located within the network, it should eventually be found.
- Due to MANET characteristics, a major requirement for our system is that it utilizes the network resources in a sensible way. A resource localization should be completed with a minimal cost.

We obviously want the resource localization to finish within a reasonable amount of time, but consider this a less important requirement than good resource utilization. We are thus willing to sacrifice localization speed in order to save network and processing resources.

5.4 General Idea

We want to exploit the characteristics of *stigmergy*, as explained in Section 3.2, to locate resources and spread information on available resources throughout the network. This is done by releasing searching agents, or *ants* from the resource-requesting nodes. Ants operate in two modes: *forward* and *backward*. Forward ants are ants still searching for a given resource, while backward ants are ants retracing the path walked by the corresponding forward ant, carrying resource location information back to the requesting node.

If we consider our MANET as a graph $G = (N, A)$, where N is the set of nodes within the MANET and A is the set of arcs defining the node connectivity, two nodes $i, j \in N$ are neighbors if there exists an arc $(i, j) \in A$. Because of node mobility, both N and A may vary over time, as nodes join and leave the MANET, and nodes move in and out of range of each other.

Forward ants perform resource localization by choosing probabilistically the next node within the graph neighborhood to which the resource request should be forwarded. Backward ants move deterministically, retracing the path traveled by the corresponding forward ant during the resource search.

Agents leave pheromone trails as they move along the graph. These trails are used to bias the probabilistic choice. As explained in Chapter 3.2, most ACO approaches use these trails to look for the *best* path from source to destination:

- Agents leave pheromone traces on their way *back* from the destination towards the source, making it possible to adjust the amount of pheromone to the goodness of the discovered path.

- A few paths, namely the ones with the highest goodness value, are used by most agents, leaving other paths more or less unused.

The result from these characteristics is that most ants follow the *same, best* path between a given pair of source and destination nodes. In our case, however, walking the same path over and over again looking for the same resource is not necessarily what we want: We assume that if two ants have initiated a search for the same resource, the second ant is searching because the first ant did not find anything. The second ant should thus look somewhere else instead. Thus, we want the opposite: To walk as many *different* paths as possible to visit a larger range of nodes and thus increasing the probability of finding an available resource that matches the search criteria. To achieve this, we will try to exploit *"opposite stigmergy"*, which in principle implies the following:

- Walk the paths with the *lowest* pheromone concentration, thus the paths that are *least recently* (if ever) walked.

A more thorough motivation for this and explanation of how we apply this is given in Section 5.5.5.

Above, we made the assumption that the only reason for two ants to look for the same resource is that the first ant did not find anything. This assumption may be somewhat unrealistic: There might be several nodes in a network looking for the same resource. Thus, there is a chance that the first ant actually found what it was looking for and thus the second ant should be allowed to locate the same resource. This problem is solved by introducing *location learning*, where backward ants leave information about located resources in intermediate nodes. Location learning is further discussed in Section 5.5.6.

Given this general idea for a solution, our approach exhibits several of the characteristics of indirect communication listed in Section 3.1:

- *Late binding of recipient:* When pheromones are left on a branch, neither the node or any other entity within the system knows if and when the trail will ever be used by other subsequent ants.

- *Anonymity:* When leaving pheromones on a branch, our ants do not know which other ants (if any) will arrive at this node later and use the trail to choose a next hop. Neither do ants leave any identification, thus, it is impossible for subsequent ants to find out who deposited the pheromone trail.

- *Time decoupling (asynchrony):* There may be an arbitrary long time interval between the time an ant leaves a pheromone trail to the time a subsequent ant arrives and uses the trail to choose a next hop.

- *Space decoupling:* Ants using pheromone trails deposited at a node do not need to have been located at the node at the time a preceding ant left the trail.

For our system, the *non-intentionality* and *analog nature* characteristics do not apply, as our artificial ant *do* leave pheromones in order to enable communication and cooperation only, and the medium of communication may not be the real world — this approach will only work for computer networks and artificial ants.

5.5 Issues

5.5.1 Scheduling

As described in Section 2.1.3, routing protocols may be either reactive or proactive. The same applies to our application: In a *proactive* approach, nodes might periodically release agents to localize resources, even though these resources aren't necessarily needed at the moment. In this case, we need to decide which resources to look for beforehand, which is a rather unrealistic assumption. Another, more realistic, proactive possibility is that resource holders periodically broadcast their current resource situation. The drawback with this approach is the same as with proactive routing protocols: One risks wasting resources on disseminating and storing location information for resources that no one will ever look for. The advantage is of course very low response time.

In a *reactive* approach, nodes release agents when a specific resource is needed. This leads to less dissemination overhead, but also a higher initial delay:

- All nodes would have to search for the resources they need themselves.
- If a resource disappears (resource holder goes down), a new search is needed.

A *combination* of these two is also possible. In this approach, the algorithm would initially be reactive, and after this phase, it would exploit proactive maintenance:

1. Nodes release agents when a specific resource is needed.
2. Nodes keep looking for the resource as long as it is needed.

This approach combines the advantages of the proactive and reactive approaches:

- Only information relevant to some node(s) is stored.

- If a resource holder goes down, we can (if the application allows it) switch to another, already known, resource holder (if there exists one) without initiating a new search.

This approach is similar to that of AntHocNet[8], which was described in Section 4.3.2.

The strictly proactive approach is ruled out, as our main requirement is low resource usage, and not low response time. Which of the two remaining approaches to use, depends somewhat on the application and the network characteristics: If resource-holding nodes may disconnect frequently, or the requested resources are unstable or perhaps may dry out, the combination approach may be considered. If node connectivity and resource presence are more constant, a pure reactive approach should be sufficient, as the cost of initiating new searches when needed will be lower than that of maintaining all resource locations.

For this first prototype we begin with the simplest approach, and have thus chosen a pure **reactive** approach for our solution. When this prototype is done and has been thoroughly tested, we may consider trying a more complex approach such as the combination approach to see how that affects the solution's performance.

5.5.2 Communication

Choice of Transport Protocol

In the TCP/IP model, the *Transport Layer* is responsible for end-to-end transfer by connecting applications to service ports. This layer is the lowest layer in the model to offer any reliability, such as flow control and congestion control. The two most common transport layer protocols are UDP and TCP.

UDP The *User Datagram Protocol* (UDP) [28] provides best-effort, end-to-end datagram delivery. Delivery is not guaranteed, no ordering is done upon arrival, and nothing is done to avoid network congestion. However, the low level of reliability makes transmission cheap, resulting in fast delivery. UDP is thus suitable for real-time systems and systems with scarce network resources.

TCP The *Transmission Control Protocol* (TCP) [29] provides reliable, ordered end-to-end stream delivery. Reliable and ordered delivery means that TCP guarantees that all sent packets will arrive correctly at the destination. If some packets are lost, they will be resent, and if packets arrive out of order, they will be reordered. TCP also includes tools for network congestion avoidance.

However, guaranteed delivery does introduce a cost: TCP is a connection-oriented protocol, using a *three-way handshake* to establish its connection.

In a three-way handshake, the client sends a synchronize packet (SYN) to the server, which replies with a synchronize-acknowledgement (SYN-ACK). Finally, the client answers with an acknowledgement (ACK), and the connection between the client and the server is established. Thus, there is an overhead for each connection that is set up. To ensure delivery, overhead is introduced in terms of *acknowledgements*, meaning an extra load on the network. Delivery might also be delayed, as one packet can not be delivered to higher layers before all previously sent packets have been delivered.

In addition to introducing a substantial amount of transmission overhead, which is something we want to avoid, TCP is extensively tuned to give good performance in traditional, wired networks [11]. When a packet is lost, TCP assumes this is due to congestion and invokes its congestion control mechanisms. However, in a MANET, the packet loss did not necessarily occur because of congestion. The mobility in MANETs often results in frequent route changes, and a route loss may lead to packet loss. Packet loss due to connectivity changes should be handled differently than congestion, and as TCP does not differentiate between these two situations, it is less suitable for use in MANETs.

By choosing UDP, we risk that some datagrams get lost, meaning that we have to re-initiate the resource localization. However, UDP is much more suitable in MANETs, and we have thus chosen **UDP** as our transport layer communication protocol in this solution.

Topology Updates

For nodes to be able to further forward incoming forward ants to one of their neighbors, they obviously need to know who their neighbors are. They also need to receive topology updates every time a node either enters or leaves their neighborhood.

This kind of topology information is kept by the routing protocol, and may be fetched from it.

Choice of Routing Protocol

In the above discussion, we decided to use a *reactive* approach to use less resources on resource localizations. Still, we have chosen the *proactive* OLSR as our routing protocol. One main reason for this is that we need up to date neighbor information at all times to make correct next hop selection possible. OLSR keeps constant control over the network topology, and is thus able to notify our application of any topology changes as soon as these are discovered. AODV, which is one of the most frequently used reactive routing protocols, does not keep track of the network's entire topology, but rather broadcasts a route request whenever a route is needed. Such a protocol is not able to provide us with the topology information that we need. Thus,

in our application, we use **OLSR** as our routing protocol.

Node Mobility

The aim of this thesis is not to design a new routing protocol. Therefore, we rely on the underlying routing functionality to handle the mobility of the nodes. Thus, we only need to know what nodes have which resources, not the actual position of these nodes. Because of this, it is sufficient that the resource holder stays within the same network partition — within the partition it may move wherever it wants, and may still be found by the nodes targeting its resources.

5.5.3 Supported Network Topology

As explained in Section 2.1.2, MANETs may be either dense or sparse. In a dense network, partitioning occurs infrequently, whereas in sparse networks, partitioning may occur more often. Developing applications for sparse MANETs is a more challenging task than developing for dense networks, as one has to consider the arising and merging of partitions. Typically, this involves keeping the system in a consistent state during partitioning and merging.

As the goal for this thesis is to see if ant-inspired algorithms are suitable for our purpose, we have chosen to restrict ourselves to **dense** networks. If the approach turns out to function well in dense networks, it should also be possible to expand to work in sparse networks. This will be further discussed in Section 9.4.1.

5.5.4 Resources

The purpose of the system is to enable nodes to search the MANET for a certain resource. A node thus needs information on its own resources in order to be able to share this kind of information with other nodes.

In our prototype, we have gone for a very simple approach where all nodes keep a resource file containing information about all present resources. For each resource the node wants to offer to other nodes in the network, the file includes one line containing the name of the resource followed by some integer quality measurement. This quality measurement enables other nodes to request not only a resource, but a resource satisfying a certain level of goodness. Alternatively, this value may function as a boolean value where a quality measurement is not suitable, telling if a certain resource is present or not.

It is assumed that all nodes have a common way of representing these quality measurements, and that this representation is known to all nodes within the network.

5.5.5 Pheromone Traces

In most ACO systems, to each arc of the graph $G = (N, A)$, a value τ_{ij} is associated, representing the pheromone trail for the link between nodes i and j.

In our case, however, the fact that different ants may be searching for different resources somewhat complicates this matter. For example, say one node has initiated a search for resource A. Then some node initiates a search for resource B. The ant searching for resource A has left pheromone traces along its path telling it has recently used this path to search for a resource. If we do not differ between different resources, this will give the ant searching for resource B a higher probability for choosing other next hop neighbors to prevent the two ants from searching along the same path. The fact that one ant did not find resource A at one node, however, does not necessarily mean that the same node does not contain resource B either. Thus, pheromone values should be assigned to *{neighbor, resource}* couples, not only neighbors. Our pheromone trails are thus denoted τ_{ijr}, where r is a given resource, and *resource* is a combination of both resource name and quality.

Pheromone Depositing

As mentioned in Section 5.4, most ACO systems try to differ between paths by their goodness, meaning that pheromones need to be deposited by backward ants after estimating the goodness of the chosen path. In our solution, we want to exploit what we called ”opposite stigmergy”, meaning that instead of walking the best paths or the most used paths, we want to try the least recently used path to get a better dissemination of our forward ants.

In principle, our ants may leave pheromones on their way back, even though the usage of our pheromone trails is somewhat different from the traditional usage. However, as already stated, we do want our searching ants to spread as much as possible. If ants should leave their pheromones on their way back, we might end up with several ants looking for the same resource along the same path: Say, two ants, $A1$ and $A2$, are looking for the same resource, resource R, located at node E in the MANET depicted in Figure 5.1. Both ants arrive at node A. The link from node A to node B is the least recently used path, and is thus probably chosen by the first arriving ant, say $A1$. If this ant does not deposit any pheromone before returning, and ant $A2$ arrives at node A before ant $A1$ returns, there is a great probability that $A2$ is also forwarded to node B, as this is, according to the pheromone values stored at node A, still the least recently used {neighbor, resource} couple. This, however, does not give a very good dissemination of the ants, which was what we really wanted. Thus, pheromone values should be deposited as soon as the link is used, thus by forward ants.

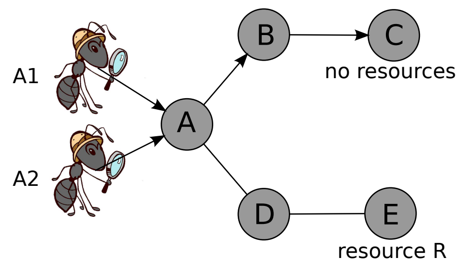


Figure 5.1: A simple MANET where depositing pheromones on the way back would give a poorer dissemination of the forward ants arriving at node A.

As mentioned above, a slight disadvantage with this approach is that it assumes that no preceding ant looking for the same resource actually found it. Every time a node initiates a resource localization and a forward ant is produced, we try to send this ant along a not recently used path, hoping this time it will actually find something. What we do not consider is the possibility that the last ant actually found a resource, meaning that all later ants should actually try the *same* path, hoping the resource is still there.

This problem is partly solved by resource location learning, as explained in Section 5.5.6. However, learned resource locations expire after a while, and after this point, a new, complete search has to be done. It might, however, be a good idea to let prior knowledge like this influence the choice of next hop a bit, as there is a possibility that the node is located nearby its old position and still has the requested resource. There should still be a possibility to tailor this according to resource and network characteristics, though: If nodes or resources tend to come and go, initializing a new search after every expiry might introduce more load than trying to locate a resource that is no longer there.

Resource Goodness

In most ACO systems, pheromone values say something about the *goodness* of a path — for example, in a routing solution, how smart is it to route packets destined for node C on the link from node A to node B? To measure such goodness, one might measure the time it took to send a message along this route and compare the results to results from routing via other neighbors. Pheromone values are then updated by the backward ants.

It might be useful to introduce goodness also in our system — perhaps one resource is located closer to the requesting node than others, or perhaps there is more left at one node than others, if we are looking for a resource that a node may run out of. This would, however, complicate our pheromone value solution, as the goodness of a {neighbor, resource} couple will not be known before the resource is actually located. This means that pheromones would have to be deposited both by forward ants as well as by backward

ants. A way of weighting and combining the different kinds of pheromone values would then also be needed to secure that none of the two had too much impact on the next hop choice relative to the other. Also, a goodness value indicates that a resource is actually present, whereas our pheromone values only indicate which paths have been used to search for the given resource. How to combine and use these two is thus not trivial.

In order to keep the solution fairly simple, we have thus chosen not to include this in our solution: If a resource satisfies the demands stated in the resource request (the forward ant), it is considered ”good enough” and returned, and the search is thus terminated.

Pheromone Initialization

To be able to assign probabilities according to when a link was last used to search for a given resource, we need a timestamp telling when each {neighbor, resource} couple was last used. As initial value, we use the time of startup for the system, thus favoring links that have never been used to look for the given resource.

However, as pheromone values are assigned to {neighbor, resource} couples, no pheromone values may be assigned before we have resources to assign them to. Thus, no initialization is done before a forward ant with a request for a not earlier seen resource is received. At this point, we assign pheromone values to all combinations of known neighbors and the new resource.

Pheromone Updates

Each time a link is used to look for a certain resource, the corresponding pheromone value is updated with a timestamp showing the time of the latest use.

Pheromone Evaporation

In real ant colonies, pheromone traces decrease in intensity over time because of evaporation. In most ACO solutions, this is simulated by applying a pheromone evaporation rule. As our pheromones are just timestamps, we get ”built in” pheromone evaporation, as the time since last use increases when a {neighbor, resource} couple is not used. No additional evaporation should thus be needed.

5.5.6 Ants

The ants are the agents that are going to search the MANET for resources. Ants should be made and sent upon request. Like in ACO routing algorithms, we differ between *forward* and *backward* ants. Forward ants are ants that are still searching for a given resource. Backward ants are ants that have

found a resource satisfying the resource demand, and are on their way back to the requesting node with information about the located resource.

Structure

The solution may be either *uniform* or *non-uniform*. The simplest case is a non-uniform model, where only one or a subset of the nodes in the MANET may send resource requests. In a uniform model, none of the nodes take on a distinguished role, thus all nodes may send resource requests. For our prototype, we see no reason why some nodes should be allowed to search for resources and some not, and our solution is thus **uniform**. All nodes are able both to send as well as receive ants, forward as well as backward.

Ant Information

Some basic information is bound to be included in the ants for them to be able to fulfill their purpose: For other nodes to know what the ant is looking for, it needs to contain some resource information. For this simple prototype, we have chosen to include only a resource name and a quality measurement.

Ant Lifetime

To prevent ants from living forever and wandering endlessly around the MANET searching for a resource that is not present, each ant is equipped with a *time to live*-value. This value tells how many hops an ant may travel. If a node receives a forward ant with time to live = 0 and the requested resource is not present within this node, the ant is discarded. The time to live-value is not used with backward ants, as these should only travel the path given in the ant. This value should be provided by the user upon startup, enabling the user to tune this value according to application and network characteristics.

Ant Memory

Ants need memory about addresses of traversed ants in order to be able to walk the same path back to the source. At each intermediate node, this node's address information should be added to the forward ant. For backward ants, each node should gradually reduce their address information to avoid transmitting larger messages than necessary.

Handling Incoming Ants

Incoming ants are handled according to their type:

Forward Ants Whenever a forward ant is received, the receiving node should read the resource information from the ant and check if the requested resource is present. If it is, a backward ant should be made and sent back towards the source according to the path included in the forward ant. If the resource is not present and the time to live is larger than zero, the forward ant should be forwarded to a new neighbor.

Backward Ants When receiving a backward ant, resource information should be extracted from the ant and stored at the node. The node should also check whether the backward ant was destined for itself or some other node. If it is a reply to the node's own request, this is reported and the search is terminated. If not, the backward ant should be forwarded towards the requesting node according to the path information included in the backward ant.

Search Termination

Ants may be lost because of transmission errors. Ants may also get "stuck" in parts of the MANET where the requested resource is not present, making the time to live reach zero and thus making the ant die. In either case, the requesting node will not receive an answer. To handle this, a new search is initiated if the requesting node does not receive an answer within a certain period of time. Like the time to live-value, the amount of time the ant should wait should be given as an argument to the program, making it possible for users to tune this value to the application needs and the network characteristics.

To prevent a search for a resource not present in the MANET from going on forever, there should also be a way to stop a search, either by providing a maximum number of initiated searches or by user input. The latter is more adjustable, but does demand user intervention. For simplicity, this has been excluded from our solution.

Choosing the Next Hop

When a forward ant arrives at a node which does not have the requested resource or any location information on the requested resource, the ant needs to be forwarded to one of the node's neighbors. The pheromone values stored at this node are used to calculate the probability of choosing each neighbor, based on the time since it was last used to look for the given resource. The longer time since last use, the larger probability to be chosen next.

A simple but reasonable way of assigning these probabilities is to look at the ratio between the elapsed time for one neighbor and the total elapsed time at all neighbors. If e_{sdr} is a measure (in seconds or milliseconds) of the elapsed time since neighbor d was used as next hop from node s to look for

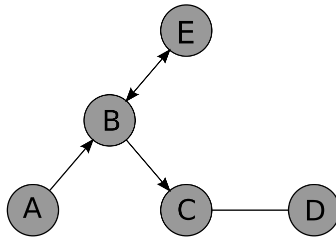


Figure 5.2: A simple MANET where allowing ants to travel in loops might increase network utilization.

resource r, then the probability p_{sdr} of creating at node s a forward ant with node d as next hop is:

$$p_{sdr} = \begin{cases} \frac{e_{sdr}}{\sum_{i=0}^{n} e_{sir}}, & \text{if } d \in \mathcal{N}_s, \\ 0, & \text{if } d \notin \mathcal{N}_s \end{cases}$$

where $\mathcal{N}_s$ is the set of neighbors of node s, excluding the forward ant's last predecessor. To help the ant discover as many new nodes as possible instead of revisiting already visited nodes, we do not allow an ant to get sent directly back to the last hop if other neighbors are available.

Handling Loops

In the sample network shown in Figure 5.2, we see that we risk that a forward ant from node A is sent via node B to node E. If we are not allowing loops, the forward ant would stop here. Terminating the search here would mean wasting a lot of resources on a search that did not find anything, even though it might have if we had let it continue. We should thus allow loops, or, in other words, allow forward ants to be sent back the same way as they came, if this is the only option.

In traditional ACO systems with pheromone disposal done by backward ants, loops are a problem because they would lead to pheromones being deposited several times at the same node by the same backward ant. In our solution with only forward ants depositing pheromones, this is not a problem. However, we still don't want backward ants to travel in loops, as this would waste unnecessary network resources.

This could be done either in the resource-holding node or by each node on the path back to the requesting node. In the former approach, the resource-holding node would remove all loops from the entire path. In the latter approach, each node would search the list of preceding nodes for their own address information and thus only remove loops concerning themselves. This approach spreads the computing on more nodes, but with the size of our target networks, this is not a costly operation and may thus just as well be

done by one single node alone, simplifying the reception of backward ants at intermediate nodes.

Thus, when transforming a forward ant into a backward ant, loops should be removed from the list of visited nodes.

Resource Location Learning

As backward ants travel back to the requesting node the same way as the forward ant came, an obvious optimization is to store information about all localized resources at all intermediate nodes. This way, subsequent forward ants may terminate their search a lot earlier if they reach a node with stored location information on the requested resource. Also, node mobility may help move information about resources in one part of the network to other parts of the same network, further improving search time in later resource localizations.

In our prototype, when receiving a backward ant, the node will store the location information carried by the ant *if* the new information is considered better than than any existing information on the same resource. At the time, this decision is taken based on the quality measurements only. However, it might be a good idea to also consider the age of the old and new information - the old information may be about to expire, or it may actually be more recent than the newly arrived information.

When a node receives a forward ant, it first checks if it has got the resource itself. If not, it checks if it has any recent information about the location of the resource. If it does, it replies with a backward ant. If not, it forwards the forward ant as usual.

MANET characteristics may make this kind of information volatile. As we rely on the underlying routing protocol to find the resource-holding node after localization, we do not have to handle intra-partition mobility. However, resource-holding nodes may disconnect or move out of the current partition, making previously reachable resources unreachable. Also, all resources may not last infinitely — some nodes may run out of certain resources after a while.

To handle this, resource location information should expire after a certain time period. This time period may be tuned to fit the network and resource characteristics. We have chosen to keep this value equal for all nodes. However, one might consider setting this individually for each node or even each resource. The information should then be included in the backward ant.

5.6 The Localization Algorithm

In this section we will summarize the above discussion through a simple pseudo-code for the final algorithm for resource localization used in our implementation and experiments. The pseudo-code is listed in Figure 5.3.

```
listen for incoming ants from neighboring nodes
listen for incoming resource requests from local
       applications
listen for incoming topology updates from olsr


if forward ant is received:
  extract requested resource information
  if resource is present:
    produce backward ant containing own
    resource information
  else:
    if resource location is known:
      produce backward ant containing known
      resource information
    else if ttl != 0:
      add own address information to the forward ant
      choose a next hop neighbor
      forward the modified ant to this neighbor

if backward ant is received:
  record new resource information
  if backward ant is reply to own resource request:
    report resource as found
  else:
    peel off own address information from ant
    forward ant according to ant memory information

if resource request is received:
  if resource location information is known:
    report resource information
  else:
    while reply is not received:
      produce new forward ant
      choose a next hop neighbor
      send ant to this neighbor
      sleep for max_search seconds

if topology update is received:
  if new neighbor:
    add node to list of neighbors
  if neighbor is lost:
    remove node from list of neighbors
```

Figure 5.3: Pseudo-code for the ant solution.

Chapter 6

Implementation

"Vision without implementation is hallucination"

— *Benjamin Franklin*

In this chapter, we will explain the implementation of the design derived in Chapter 5. The focus will be on the same parts of the program as those explained in the design. As C code tends to get long even when doing quite simple things, we will not show the complete code for the program, but rather code snippets when appropriate. Some of the most important functions are listed in Appendix D.

The first sections address some general issues such as choice of programming language and emulator-specific adaptations, before we move on to the more specific parts of the program and the various data structures used in Sections 6.6 through 6.11. Finally, Section 6.12 provides an overview of the program flow and gives program execution instructions.

6.1 Remarks

This implementation is regarded as a prototype only. Our focus has been on the ant principles rather than the code itself. It should thus be noted that quite a few simplifications have been made during the implementation. At several stages other, more elegant or resource utilizing approaches probably exist. These considerations have, however, been left to a later stage, if the solution turns out feasible for its purposes.

Also, even if this application is meant to run within a MANET, the code is not written with this particularly in mind. Thus, in a production implementation, changes should be made to the code to make it more suitable for MANETs and their characteristics. For instance, messages sent over the wireless link should utilize tailored data types to use as little bandwidth as

possible. Also, one might want to tailor other data structures and computations to use as little computing resources as possible.

6.2 Programming Language

We have chosen to implement the prototype using the C [16] language. C is a quite small and quite low level, general-purpose programming language [21]. C is not tied to any specific hardware or system, which makes C code highly portable. In addition to this, C provides low-level memory access as well as easy but powerful socket and networking facilities, making it especially suitable for our solution.

Also, we are going to do our testing with the NEMAN emulator. When running emulations, using a lightweight language such as C may be preferred over other, more heavyweight languages such as Java, which requires a virtual machine and thus needs more computing resources per virtual node, as this enables us to run tests with more nodes on one single computer.

6.3 Developing for NEMAN

Emulating in general and with NEMAN specifically will be explained in Chapter 7. We do already state, however, that emulator testing requires very little program code adjustments. The only required adjustment in our code is that for the application to be able to listen and send messages only on a specified virtual device, the application, or more specifically an application's sockets, need to be bound to this particular virtual device using the SO_BINDTODEVICE option. This is needed because all virtual devices are running on the same machine.

More specifically, this is done in our code by calling the function *bind_socket_to_device()* with a socket descriptor as argument. This function is shown in Figure 6.1.

When using NEMAN, a message sent via one node to another might be delivered up to the application layer at the intermediate node, even though it was not actually destined for this node. This behavior is normally not seen in real life, but to make everything work with NEMAN, we include information about the actual destination in all messages to make sure no one handles messages meant for other nodes.

6.4 The sockaddr_in structure

The **struct sockaddr_in** is a builtin C structure that is heavily used in our implementation. This structure is used to address remote endpoints, and it is the structure used with the *sendto()* and *recvfrom()* system calls used in our implementation for sending and receiving messages, respectively.

```
int bind_socket_to_device(int sockfd)
{
  int rv;
  if((rv = setsockopt(sockfd,
                      SOL_SOCKET,
                      SO_BINDTODEVICE,
                      iface,
                      strlen(iface)+1)) == -1)
    {
        write_log_error("so_bindtodevice");
    }
    return rv;
}
```

Figure 6.1: C code for binding a socket to a specific device.

```
struct sockaddr_in {
  short           sin_family;
  unsigned short  sin_port;
  struct in_addr  sin_addr;
  char            sin_zero[8];
};
```

Figure 6.2: sockaddr_in C structure.

The contents of this structure is shown in Figure 6.2. The fields **sin_port** and **sin_addr** hold the port number and IPv4 address of the remote node respectively.

6.5 Logging

To maintain a certain structure on the program output, we have chosen to keep two kinds of logs: One for debugging purposes and one for system tracking. The debug log is used to ease debugging, and is written to whenever something unexpected happens within the system. To ease error searching, we keep only one ”global” file. Obviously, this file will only be global if all devices are actually running on the same machine, as is the case when testing with NEMAN. If not, this file will also be local, as the path of the file will be local to the machine the program instance is running on. In either case, each line in this file starts with the name of the node reporting the error. This file is called *<program name>.debug*, e.g. *ant.debug*.

For system tracking we keep one log per node. These files contain infor-

mation on what has happened on each node, e.g. what forward and backward ants have been received, where they have been forwarded and what resource locations have been learned. These files are called *<interface name>.log*, e.g. *tap1.log*.

When running tests, some additional information is needed. When running Make with the target *test*, an additional file, *ant_test.log*, is made. All nodes write to this file whenever a request for a present resource is received.

6.6 Neighbor Information

6.6.1 Topology Information Retrieval

Neighbor information is retrieved from the OLSR daemon running on each device. More specifically, this is done by establishing a TCP connection to **localhost** on port **1212**, which is listened to by OLSR. After connecting, OLSR will report any topology changes to port number **3458**. OLSR messages are on the following form:

```
<new/old route>,<to IP>,<via IP>,<no. hops>,<from device>
```

For example, the following message:

```
1,10.0.0.2,10.0.0.2,1,tap1
```

means that there is a new route from node tap1 to the node with IP address 10.0.0.2 via the node with IP address 10.0.0.2. This route is one hop long, which means that 10.0.0.2 is a neighbor of tap1.

If the message starts with a 0, this means that an old route is lost. The following message:

```
0,10.0.0.2,10.0.0.2,1,tap1
```

means that the old route between tap1 and 10.0.0.2 is lost.

However, if a neighbor moves out of range, but there still exists an alternative path via other nodes to this node, OLSR will report this as a new route, not a lost route:

```
1,10.0.0.2,10.0.0.3,4,tap1
```

In this example, there is still a route from tap1 to 10.0.0.2, but the new route goes via 10.0.0.3, and is 4 hops long. Thus, if a new route message is received but the number of hops to this node is more than one, this node should be removed from the list of neighbors (if present).

```
struct neighbor {
  struct sockaddr_in *si_n;
  struct dl_list *pheromones;
};
```

Figure 6.3: Neighbor structure.

6.6.2 Neighbor Registry

Each node keeps a doubly linked list, stored as a global variable called **neighbors**, of **neighbor** structures, shown in Figure 6.3. This list is altered as new topology information is retrieved from OLSR. New neighbor information is always inserted at the beginning of the list. If a node loses one of its neighbors, the entry is removed from the list.

The **si_n** is a sockaddr_in structure containing the address information of the neighbor. The list pointer **pheromones** is a pointer to a list of pheromone information associated with this particular neighbor. This data structure will be further explained in Section 6.8.1.

6.6.3 Topology Changes During Resource Localization

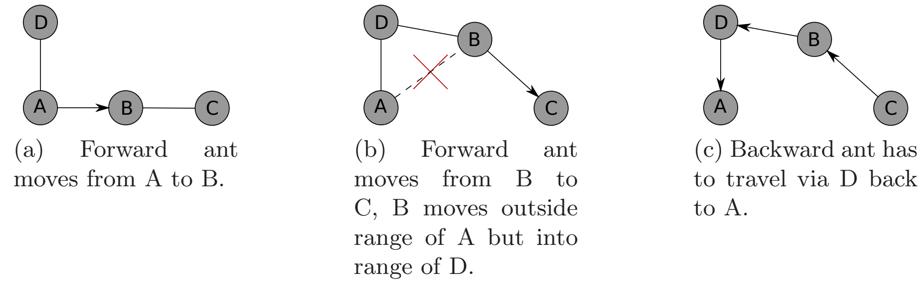
Because of mobility and nodes joining and leaving the network, topology changes may occur during a resource localization. This leads to some issues that need to be handled by our system.

Lost Neighbors

Neighbors may be lost between the forwarding of the forward ant and the reception of the corresponding backward ant, as shown in Figure 6.4. In Figure 6.4a, node A first forwards a forward ant to B. In Figure Figure 6.4b, B forwards the ant to C and moves out of range of A before the corresponding backward ant arrives. When the backward ant arrives at B in Figure 6.4c, A and B are no longer neighbors, and the backward ant needs to be routed via D to A.

We could check this when forwarding a backward ant, but we have again chosen to rely on the routing protocol. As long as the backward ant is received at the requesting node, it is not crucial that all the intermediate nodes are actually neighbors, as no pheromones are left by backward ants. They do still, however, need to be located in the same partition for this forwarding to be possible. If they are not neighbors, we may actually get a better dissemination of resource locations throughout the MANET.

With this approach, we do risk that an old neighbor has moved out of the partition or simply gone offline, making it impossible to forward the backward ant to the requesting node. However, there might be other paths



(a) Forward ant moves from A to B.

(b) Forward ant moves from B to C, B moves outside range of A but into range of D.

(c) Backward ant has to travel via D back to A.

Figure 6.4: Neighbor is lost between forwarding of forward ant and reception of corresponding backward ant. Arrows show where the ants are moving.

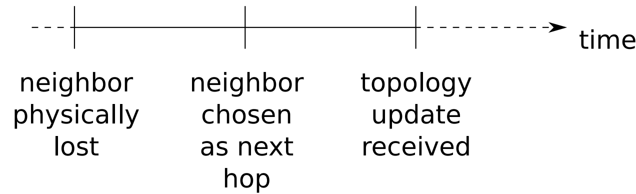


Figure 6.5: Neighbor is chosen as next hop before topology update is received by the ant application.

to this node. Thus, it might be a good idea to, in addition to the backward ant, send a request reply directly from the resource-holding node to the requesting node, increasing the possibility that the requesting node actually gets a reply. This would introduce a slight overhead, but might be worth it if the node mobility is high enough. However, this has been left as further work.

Delayed Topology Updates

Topology updates are delivered over a TCP connection from the OLSR daemon. There will always be a slight delay from the actual topology change occurs to the topology update information is delivered from OLSR to the ant application. Information on *new* routes only makes the new neighbor impossible to select as a next hop until the topology update is received, and should thus not cause any problems. Delayed route *loss* updates, however, make it possible to choose as next hop a node that is no longer a neighbor, as shown in Figure 6.5.

The chances of choosing a lost next hop neighbor during this very short interval is of course very small, but it might still happen and should thus be handled. However, as we are depending on the underlying routing protocol, a message sent to a lost neighbor should be received at this node as long as there is an alternative route to this node. Only if the node is completely outside the partition will the ant not be delivered there. This will make

```
cpu 50
memory 70
printer 1
```

Figure 6.6: A sample resource file.

the ant be sent via other nodes towards this ”neighbor”, but this should not really make a big difference for the localization. As soon as the route loss information is received, all pheromone information for this node will be deleted from the system, and this somewhat wrong forwarding decision will thus not make any long lasting influence on the rest of the system.

6.7 Resources

6.7.1 Local Resource Information

At each node, resource information is stored in a file called *<iface>.res*, where *iface* is the name of the node's interface, e.g. *tap1.res*. This file contains one line per local, publicly available resource, each stating the resource name and quality, as shown in Figure 6.6. The ”printer” quality measurement may be an example of a boolean value - either a printer is present or it is not. Alternatively, it may say something about what kind of services the printer is offering. The important thing here is thus that nodes in the MANET have a common understanding of these quality measurements.

This file is read each time a resource request is received. An obvious improvement would be to keep this kind of information in memory. This would require a way of detecting updates to this file, or at least reading the file periodically to fetch updates. This improvement has been left as further work.

6.7.2 Resource Sharing

When transmitting and storing resource information a **resource** structure, shown in Figure 6.7, is used. This structure contains the information found in the .res-file; The name of the resource and a quality measurement. The resource name may be maximum RES_SIZE bytes long. RES_SIZE is a macro, currently set to 50.

The resource structure also contains a field **resource_holder**. This field contains the address information for the resource-holding node, and is used by all nodes receiving the resource location information through a backward ant, both the resource-requesting node as well as any intermediate nodes.

```
struct resource {
  char name[RES_SIZE];
  int quality;
  struct sockaddr_in resource_holder;
};
```

Figure 6.7: Resource structure.

```
struct resource_info {
    struct resource resource;
    time_t expiry_time;
};
```

Figure 6.8: Resource info structure.

6.7.3 Resource Location Info

Gathered resource location information is stored in **resource_info** structures, as shown in Figure 6.8. Each node keeps a doubly linked list, **known_best_resources**, containing all gathered resource location information. A resource info structure contains a resource structure, as explained in section 6.7.2, as well as an expiry time.

The **expiry_time** entry tells how long the snooped resource information is valid, and should be adjusted according to the network characteristics. At the time, this value is equal for all nodes. Alternatively, it might be set by the node holding the resource.

At the time, only the *best* known resources are stored, meaning that we only store *one location per resource name*. This eases resource knowledge lookup and minimizes the storage needed. If a new resource is found, its quality is compared to the quality of any previously found resource with the same name. If the quality is better, the old entry is replaced with the new. If not, the new resource location is not recorded.

Unfortunately, for simplicity, entries are only removed from this list if the node receives a forward ant looking for the resource and the entry has expired. There is no automatic removal upon expiry. This may lead to very long lists and makes the solution less scalable, but for the networks it is supposed to handle, it should work fairly good. This could be handled by a "garbage collector" thread that periodically ran through the list and removed expired entries. However, this has been left as further work.

6.8 Pheromone Traces

6.8.1 Pheromone Data Structure

Pheromone traces are recorded for each {neighbor, resource} couple. In other words, we need to store information for each resource and each neighbor. To be able to do this, we need to record all possible resources. As all possible resources are not known, we restrict ourselves to all resources ever heard of by the node. In addition to the list of known *best* resources, each node keeps a list **all_known_resources**, containing all resources the node has ever heard about. Unlike the list of best resources, this list is add only — there is no expiry time and no entries are ever removed from this list.

The easiest approach, which is also used by many other ant systems, would be to use a two-dimensional array with neighbors along one dimension and resources along the other. However, both the number of neighbors and the number of resources which someone will look for are unknown, making it harder to declare a suitable array, although possible with dynamically allocated arrays. Also, there is no obvious mapping from resource names and node identifiers to array indices.

Another possibility is to use ”two-dimensional linked lists”. This approach, however, introduces a lot of redundancy and would require a lot of work to update.

Better solutions are possible, for example to use some kind of hash map, where the keys would be the {neighbor, resource} couple and the values would be the pheromone values. Although independent C libraries for hashing are available, we have chosen to use the list approach, as this introduced less implementation issues.

When using the two-dimensional list approach, we need to decide what information should be stored on which list dimension. If the first level list is the neighbors and the second level is one resource list per neighbor, we only have to register each neighbor once, which is good. Adding and removing neighbors from this list is easy and cheap. Resources don't need to be removed from the lists, so we only need to add these, even though they would need to be added to every single neighbor when they first appear. However, summarizing elapsed times for each neighbor for each resource, which is needed when choosing the next hop for a forward ant, would in the worst case require looping over the entire list structure.

On the other hand, if the first level list is a list of all resources, we have to register all neighbors for all resources. As neighbors might come and go, this might require us to update each of these lists quite frequently, both adding and deleting entries. Summarizing would be a lot easier, however, as we only need to loop through the list of neighbors for the requested resource and sum these, thus we only need to loop through one list, not the entire two-dimensional data structure.

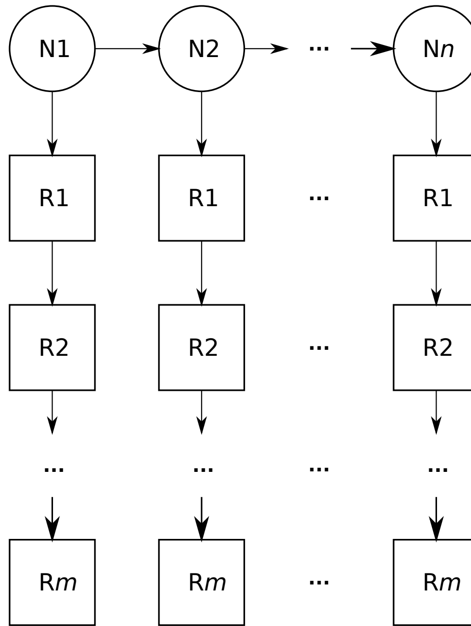


Figure 6.9: Two-dimensional linked list structure. First level list consists of n neighbors, second level list consists of m resources.

```
struct pheromone {
  struct resource *resource;
  time_t last_used;
};
```

Figure 6.10: Pheromone structure.

Which approach to choose depends on the usage. What happens most frequently: Neighbor updates or resource requests? If nodes move around a lot, this implies a large amount of updates to the neighbor lists, and it will thus be good to keep only one neighbor list. If resource requests are more frequent, however, it will be more beneficial to keep only one resource list. For this system, neighbor updates will probably occur more frequently than resource requests. Thus, we have chosen the former approach. The principles of the data structure is shown in Figure 6.9.

As the first level list is the list of neighbors, we may use the existing list of neighbors also for pheromone value storage. As explained in Section 6.6.2, each neighbor structure includes a pointer to a list of pheromones. More specifically, this is a pointer to a list of **pheromone** structures, as shown in Figure 6.10.

This structure keeps a pointer to the corresponding entry in the

all_known_resources list and the actual "pheromone": A timestamp showing when this neighbor was last used to look for this particular resource.

6.8.2 Pheromone Initialization

Whenever a new neighbor is discovered, it is added to the list of neighbors and a new pheromone list is created and added to the neighbor data structure. The pheromone list contains all the resources from the **all_known_resources** list. The last_used entry is initialized to the boot time of the application on this node, giving all neighbors the same probability of being chosen if none are previously used, and a higher probability of choosing unused neighbors than previously used neighbors if some are used and some are unused.

Whenever a node receives a resource request for a resource that is not stored in the **all_known_resources** list, it is added to this list and to all existing neighbor-pheromone lists. The **last_used** entry is again initialized to the application boot time.

6.8.3 Pheromone Updates

Whenever a node is used as next hop to search for a given resource, the corresponding {neighbor, resource} **last_used** entry is set to the time of usage.

At the time, if a neighbor is lost, the entire neighbor and its associated pheromone list is deleted. It might perhaps be a good idea to keep these lists, at least for a little while, in case the neighbor moves back into range. In systems with high mobility, nodes may move in and out of range of each other quite frequently, and thus cause frequent building of new pheromone lists, as well as re-initializing all timestamps associated with each {neighbor, resource pair}.

6.9 Ants and Ant Memory

Both forward as well as backward ants consist of an **ant** structure followed by a number of **struct sockaddr_in** structures. These structures were chosen to ease the forwarding of backward ants, as these may be used unchanged in calls to *sendto()*, the function used for sending messages. The address information is added by the receiving node, not the sending node. There are two advantages with this approach: The ant size is increased one step later into the procedure, putting less load on the network, and the address information is easily available as this is returned by the call to *recvfrom()*. The C struct used for ants is shown in Figure 6.11. The **type** field must be either 'F', for forward ants, or 'B', for backward ants. **preceding_ants** is the number of ants that have previously been sent to look for the same resource. This is

```
struct ant {
  char   type;
  int    preceding_ants;
  int    preceding_nodes;
  short  ttl;
  struct sockaddr_in requesting_node;
  struct sockaddr_in next_hop;
  struct resource res;
  struct timeval time_of_request;
};
```

Figure 6.11: Ant structure.

needed for testing purposes only, and is further explained in Section 7.6.1. **preceding_nodes** gives the number of already visited nodes, and thus the number of address structures included in the ant. **ttl** tells how many hops the forward ant has got left before it has to terminate its search, and is decremented by one for each hop the ant travels. **requesting_node** contains the address information for the node that requested the node. **next_hop** is included because of NEMAN behavior as explained in Section 6.3. For forward ants, the **res** field carries information for the requested resource, whereas for backward ants, it holds information about the located resource. Thus, the name field should be equal in corresponding forward and backward ants, whereas the quality value in a backward ant may be equal or higher than that in the forward ant. Finally, **time_of_request** is used for timing requests when doing performance testing, as will be explained in Chapter 7.

The same structure is at the time used for both forward and backward ants. Backward ants could probably be slightly smaller, as they shouldn't need the "time to live" field — they strictly follow the path copied from the forward ant. Also, in future, more sophisticated versions, other differences may reveal, making it more desireable to differ between the two kinds of ants. In this prototype we have, for simplicity, chosen to use the same structure for all ants.

6.9.1 Choosing the Next Hop

When choosing the next hop for a forward ant, we want the probability for choosing a neighbor to increase with the time since it was last used to search for the given resource, as explained in Section 5.5.6. If e_{sdr} is a measure of the elapsed time since neighbor d was used as next hop from node s to look for resource r, then the probability of creating at node s a forward ant with node d as next hop is:

```
struct neighbor_prob {
  struct neighbor *neighbor;
  struct pheromone *pheromone;
};
```

Figure 6.12: Neighbor_prob structure.

$$p_{sdr} = \begin{cases} \frac{e_{sdr}}{\sum_{i=0}^{n} e_{sir}}, & \text{if } d \in \mathcal{N}_s, \\ 0, & \text{if } d \notin \mathcal{N}_s \end{cases}$$

where $\mathcal{N}_s$ is the set of neighbors of node s, excluding the forward ant's last predecessor.

The actual choosing is implemented by generating a random number in the interval $[0, 1]$. Let's say we have a list of all neighbors and their associated probabilities for the given resource, and a variable *sum*. Then we loop through the list of neighbors, and for each neighbor add the probability associated with the neighbor to *sum*. Then, we check if *sum* is less than or equal to the random number. If it is, we choose this neighbor, if not, we proceed to the next neighbor in the list.

To do this, we use the **pheromone** data structure explained in section 6.8.1. This structure contains the time of last use for each {neighbor, resource} pair available. Because of the choice of primary (neighbor) and secondary (resource) lists in this two-dimensional list structure, this ended up getting a bit more ugly that we had hoped: To find all available neighbors and the associated time of last use for the requested resource, we have to loop through the entire two-dimensional list structure. To avoid looping over this entire structure more than once, we build a new temporary structure containing only the *appropriate* {neighbor, resource} pairs, where the appropriate pairs are those with same resource name and quality. The appropriate pairs are stored in a list of **neighbor_prob** structures, each representing a neighbor-resource pair. This structure is shown in Figure 6.12. The temporary list is deleted when the next hop has been chosen.

The entire pheromone struct is included instead of just the resource, as the time of last usage, which is stored in the pheromone structure, needs to be updated if the neighbor is chosen.

When building the list of appropriate {neighbor, resource} couples, the last hop neighbor from which the ant arrived is skipped, and thus not added to the list of available neighbors. This is done to avoid sending the request back to this node. However, if the list of available nodes is empty after this phase, meaning that this last hop node is the only neighbor, the request is sent back to this node. The alternative would be throwing the ant away, but by doing this, we allow the ant to try another path instead.

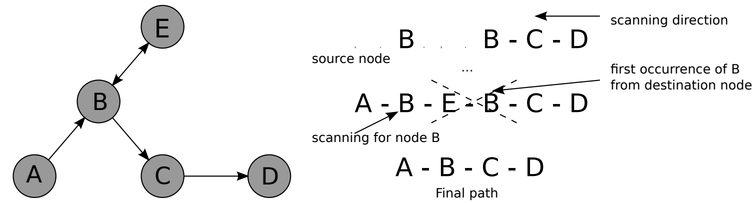


Figure 6.13: Loop elimination in backward ants.

6.9.2 Loop Elimination

As backward ants should not travel in loops, all loops are removed from the backward ant before it is sent from the resource-holding node. This is done by looping through the list of visited nodes starting at the source node, and for each node, a reversed search through the same list is used to look for equal entries. If one is found, every node in between is removed from the list. The fundamentals of the approach are shown in Figure 6.13. In this figure, the forward ant has visited node B twice. During loop elimination, the loop, i.e. nodes B and node E, is removed from the list of visited nodes.

This approach also works if the same node is visited more than twice, as the occurrences in between first and last visit are removed the first time a loop is discovered.

The number of preceding nodes is updated correspondingly to match the length of the path that the backward ant is going to walk back to the requesting node.

6.9.3 Ant Communication

All nodes are listening for incoming ants on port **3456**.

6.10 Resource Localization

In this section, we will explain how a resource localization is initiated and terminated.

6.10.1 Search Initiation

All nodes listen for incoming resource requests. Resource requests are sent to a node via UDP segments to port **3457**. Resource requests contain information about the resource, and should be on the form "<resource name> <minimum quality>". We do not check that resource requests are sent from the same device as the one that received the request, thus, anyone may send a request for a resource to any node, provided there is a path from the actual requesting node. The result, however, is only written to file on the node that received the request.

Before a forward ant is produced and sent, we check if a suitable resource localization is already known and listed in **known_best_resources**. If it is, it is not necessary to do a search. This check is performed again every time a new forward ant is about to be sent, as new resource information may have been snooped since the last ant was sent **max_sleep_time** seconds ago.

When a search is initiated, the requested resource is added to a list of ongoing searches, **not_found_resources**.

6.10.2 Search Termination

A search is terminated when the requested resource is found. While the resource is not located, a new search for the same resource is initiated every **max_sleep_time** seconds, a value which is provided by the user upon startup. Currently, there is no possibility to terminate the search in any other way.

When a backward ant arrives at the requesting node having found a proper resource, the resource is removed from the list of ongoing searches. When doing this, all nodes satisfied by the localized resource is removed.

After **max_sleep_time** seconds, we check if the list of ongoing searches still includes the requested resource. If it does, the resource was not found, and we initiate a new search. If not, the resource was found and the search is terminated.

When a backward ant arrives at the resource-requesting node, the requested resource is removed from **not_found_resources**. After each search has had **max_sleep_time** seconds to terminate, we check if the resource is still listed in **not_found_resources**. If it is, the resource is not yet found, and a new search is initiated. If it has been removed from the list, this means that the resource has been localized, and the search is terminated.

6.11 Utilities

As most of the data structures are doubly linked lists, we have made a ”generic” doubly linked list implementation used to handle all these lists. The structure for the lists is shown in Figure 6.14.

Each list element is thus of this data type. The void pointer **entry** is a pointer to the actual entry. This may for example be a neighbor struct or a resource struct. This approach introduces an extra step to access the actual information stored in the list, but makes it possible to use the same functions for adding and deleting entries in lists of any data type.

Along with the structure, we have implemented functions for inserting an element at the beginning of the list, deleting a specified element and deleting (with freeing of memory) an entire list.

```
struct dl_list {
  struct dl_list *prev;
  struct dl_list *next;
  void *entry;
};
```

Figure 6.14: Doubly linked list structure.

6.12 Program Flow

6.12.1 Threads

As the program should be able to manage several tasks at once, it is divided into several threads:

- *The main process:* Listens for incoming ants, forward as well as backward.
- *Thread 1:* Listens for incoming topology updates and handles the list of neighbors. Created by the main process shortly after startup.
- *Thread 2:* Listens for incoming resource requests. Created by the main process shortly after startup.
- *Thread 3:* Initiates a new search every *max_sleep_time* seconds. Created by Thread 2 for every incoming resource request.
- *Thread 4 and 5:* Handles backward and forward ants, respectively. Created by the main process whenever receiving an ant.

Chapter 7

Test Setup

"Experience is the worst teacher. It always gives the test first and the instruction afterward."

— *Source unknown*

In this chapter we explain the setup used during our tests, including explanations of the tools used, the network scenarios in which the tests are performed as well as the setup of the actual experiments done.

When performing a performance analysis, several choices must be made: Which evaluation technique to use, which environment to perform tests in and which metrics to evaluate the system on. This chapter gives a general overview of some of the available techniques and tools, as well as an explanation of the specific tests used to analyze our system.

When testing our resource localization solution, we need to perform tests that reflect the goal and requirements for our solution, stated in Chapter 5. The tests should verify that the goals for the solution are reached, thus that nodes looking for a resource are able to locate this resource if it is present in the MANET. We also test how well the solution performs with regard to the requirements stated in Section 5.3, thus how well the solution utilizes the resources present in the network, both in terms of networking resources and local processing power at each node. We also take a look at how fast the solution is able to localize a given resource. An explanation of how our tests are set up and performed in provided in Sections 7.5 through 7.9.

In Section 7.4, we present an alternative to the ant solution, used to get comparable results for our analysis.

7.1 Evaluation Techniques

Evaluation techniques may be split into three different approaches [17]:

- *Analytical modeling:* Involves mathematical models and formal proofs, and requires simplifications and assumptions to make usable models. Gives quite low accuracy, but requires relatively little time, given that you are trained in making such models, and also does not require the system to actually exist when ”testing”.

- *Simulation:* The system is tested within a controlled and, probably, somewhat simplified environment. This gives a higher accuracy than modeling, but requires more time and resources to perform. One may test only parts of the system, making it possible to perform tests before a complete running system is made.

- *Measurements:* Measurements means doing real-world testing, i.e. to test the system in its natural environment. Performing measurements requires the system in question to actually exist and be ready for full testing. Measurements also require a full testing environment, resulting in a high cost and low repeatability. However, as the system is tested in the environment it is supposed to run in, this technique may give very reliable and accurate results. However, environmental parameters such as system configuration, workload and time of the measurement may affect the results. The accuracy may thus vary between none and high [17].

Analytical modeling requires substantial knowledge in areas like mathematics and queuing theory, as well as the time to develop good models. This, along with the potentially low level of accuracy, makes this technique unsuitable for our testing. As real-world measurements require a lot of time and resources and provides low repeatability, also this solution is rather unsuitable for our purposes. Simulation, on the other hand, both gives moderate accuracy and takes relatively short time to perform, and is thus the evaluation technique chosen to perform our testing.

The techniques listed above may also be used in combination. For example, one might use analytical modeling together with simulation to verify and validate the results obtained from each technique. In [17, p. 32], Jain states the *three rules of validation*:

- ”Do not trust the results of a simulation model until they have been validated by analytical modeling or measurements.”

- ”Do not trust the results of an analytical model until they have been validated by a simulation model or measurements.”

- ”Do not trust the results of a measurement until they have been validated by simulation or analytical modeling.”

According to these rules, we should thus not rely only on the results from our simulations, but have them validated by either analytical modeling or

real-world measurements. However, this has, because of time issues, been left as further work.

7.2 Testing Environment

7.2.1 Simulation vs Emulation

Above, we used the word ”simulation” for testing a system within a controlled and simplified environment. However, this kind of ”simulation” may be further divided into *simulation* and *emulation*. One goal of *simulators*, such as GloMoSim [43] and ns-2 [40], is to give a detailed representation of the physical layer [31]. Simulators are suitable for large-scale networks, and provide reproducible results [30]. However, code written for a simulator needs to be rewritten before it can be used on a real platform. Also, simulators take a substantial amount of time to learn to use and program.

In emulators, such as NEMAN [31] and MobiNet [22], parts of the protocol stack may be simulated while the rest is the real implementation in question. This code requires very little adjustments to work on a real platform, possibly saving a lot of time during implementing. Like simulators, emulators produce reproducible results. However, emulators tend to provide less scalability than simulators, although this depends on the emulator implementation.

7.2.2 ns-2

ns-2 [40] is a network *simulator* which for a long time has been used to perform simulations in the ad-hoc network research area. When using ns-2, the user specifies a network topology and the movement patterns of the nodes within this topology, called a scenario. The scenario file provides a detailed description of all the nodes in the network and their behavior during the emulation, for example their start position, their distance to all other nodes in the network and in which direction they are moving at what speed at different times during the emulation. The simulator is able to ”play” this scenario, enabling the user to perform measurements and analyses of implemented applications without having to set up a real life test scenario, which may be a costly and highly time consuming operation.

7.2.3 NEMAN

NEMAN is a network *emulator*. Because of the availability of researchers with high expertise on NEMAN within the research group, we have chosen to use NEMAN to perform the performance analysis for this thesis.

For NEMAN to be able to emulate a given network topology, this topology needs to be specified in a *scenario file*, just like ns-2. NEMAN uses

standard ns-2 scenario files, which enables us to use any available ns-2 scenario file generator to make scenario files for NEMAN. NEMAN scenario files also allow scenario files to define messages to be sent to specific nodes at certain times during the emulation. This may for example be used to activate certain functionality within the applications.

NEMAN Components

NEMAN consists of three elements [31]:

- *User processes:* The applications and protocols that are to be tested.
- *Topology manager:* Manages virtual network interfaces and performs packet switching according to the topology information at a certain moment in time.
- *Graphical user interface:* Provides a graphical interface to the emulator. Visualizes the emulated network, i.e. nodes, node ranges and node movement. Induces topology information to the topology manager.

The user processes hook to virtual Ethernet network devices called TAP devices. These may be used together with the classical socket API, enabling user processes to send and receive data. However, all user processes must bind to a specific TAP device using the option SO_BINDTODEVICE to avoid interference with traffic addressed to other processes.

The virtual interface *tap0* is reserved as *monitoring channel*. This interface has an open bidirectional connection to all other virtual interfaces, and thus hears all traffic in the emulated network.

7.2.4 OLSR daemon

The OLSR daemon *olsrd* [1] is an implementation of OLSR. olsrd supports mesh routing for any network equipment with a wifi card with ad-hoc support, as well as all ethernet devices. In our experiments, we have used olsrd version 0.4.10.

7.3 Emulation and Analysis Tools

7.3.1 setdest

The scenario files needed by NEMAN and ns-2 may be created by hand, but there also exists programs that, based on some simple parameters, create ready-to-use scenario files for you. *setdest* [15] is an example of such a program.

setdest exists in two different versions, the original 1999 CMU version (version 1) and the modified 2003 U.Michigan version (version 2). For our

experiments, we will be using the original version. This version takes seven parameters:

```
./setdest -v <1> -n <nodes> -p <pause time> -M <max speed>
          -t <simulation time> -x <max X> -y <max Y>
```

This command creates a scenario consisting of *nodes* nodes, moving with a maximum speed of *max speed* meters/second for a time period of length *simulation time* seconds, initially placed within the area defined by the *max X* and *max Y* coordinates. *setdest* nodes move according to the *random waypoint model* (RWM) [18]. In RWM, nodes move between *waypoints*, which are uniformly distributed over the given area. Nodes may pause for up to *pause time* seconds at each waypoint.

7.3.2 tcpdump

tcpdump [2] is a tool for intercepting and displaying the network traffic on a specified network interface. The output from tcpdump is a description of all captured packets. This description may, amongst other things, contain information about source and destination of all packages, tcpdump uses the libpcap library to capture all packets.

Running tcpdump with the **-w** flag produces a file containing all packet data for later analysis. Our tests will be run with this option, enabling us to use other analysis tools on the data on a later stage.

7.4 A Flooding Solution

The results from the ant solution alone are not enough to say anything about the quality of the solution. Thus, we need some other system to compare the performance of the ant solution with.

As a comparative solution, we have implemented a simple *flooding solution*. In this solution, instead of sending forward ants, a resource-requesting node *broadcasts* a resource request. Each receiving node re-broadcasts the request if the resource is not present. If the resource is present, a resource reply is sent directly back to the requesting node.

7.4.1 Issues with Flooding

Flooding is a very efficient solution where one wants quick dissemination of messages into the network. In our case, we want to find a certain resource that might be located at some unknown node in the network. By flooding a resource request, all nodes in the network (partition) will get the request in a minimal number of hops, and if a resource-holding node exists, it will be able to reply quickly, thus minimizing the response time.

However, there are also issues with flooding, especially within MANETs. In a MANET, communication is performed with the CSMA/CA scheme, as explained in Section 2.1.2. In such a network, flooding comes with several drawbacks [24]. First of all, not only the resource-holding node receives the request. So does also all other nodes in the entire network. In the most basic flooding approach, every node re-broadcasts every single request if the resource is not present at the node. As broadcast messages will go back and forth between neighboring nodes, this will lead to endless flooding, again leading to massive load on the network if the request is not at some point stopped, for example after a given number of hops.

The same broadcast message will also be picked up by all neighbors at approximately the same time. Those neighbors that do not hold the resource will decide to re-broadcast, again approximately at the same time. Chances are, at least some of these neighbors are within range of each other. Their re-broadcasts will then most likely contend with each other. The re-broadcast messages may also collide with each other.

A ”massive load” is not wanted in any network, and certainly not in a MANET. A lot of adjustments may be made to the basic flooding approach to make it less resource demanding. An obvious improvement, which addresses the re-broadcasting problem mentioned above, is to only allow re-broadcasting of a certain message once at each node. To make this possible, some sort of message identification is needed, together with registration of which messages have already been re-broadcast.

Other mechanisms are also available, such as the probabilistic scheme, counter-based scheme, distance-based scheme, location-based scheme and cluster-based scheme. Generally, these schemes use different techniques to reduce the number of redundant re-broadcasts, and thus also contention and collision. For a review of these schemes, see [24].

7.4.2 Flooded Requests

A node looking for a resource broadcasts a request, currently to its one hop neighbors (IP address 10.0.255.255). All receiving nodes check if they have the resource. If they do, they reply to the requesting node. If not, they re-broadcast the request to all their one-hop neighbors.

In our simple flooding solution, we have used only one load-reducing technique, namely the one stopping nodes from re-broadcasting the same resource request more than once. In the ant solution, we kept a maximum number of hops to keep requests from living ”forever” in case the requested resource is not present within the network. In the flooding solution, however, these requests will be stopped by the former technique when all nodes have received the request once. Thus, such a time-to-live value is not used in the flooding solution.

7.5 Monitoring

A *monitor* is a tool that is used to observe the activities within a system. They observe system performance, collect performance statistics, analyze data and display results, and may also identify problem areas and suggest remedies [17]. Monitoring is a key step in measuring system performance, as this is the phase where all data is collected.

Monitors may be placed *inside* or *"outside"* the system in question. To be able to place a monitor inside a system, we need to have access to and be able to modify the system's source code. This approach may give a very high level of detail, but may also add additional overhead to the system in question, as the system is modified to make room for the monitor.

A monitor placed outside the system does (probably) not get the same level of detail, as the only data available to the monitor is the system input and output. However, no additional overhead is added to the system in question, giving a high level of result correctness.

In our testing, we use both kinds of monitors: Response time monitoring is implemented in the source code, and is inside the system. Resource utilization monitoring, however, is performed outside the system.

7.6 Metrics

To be able to measure the performance of our system, we must define a set of *metrics*. The metrics are the performance criteria that the system will be tested upon.

The outcome of a system request may be one out of three [17]: The service is performed *correctly*, *incorrectly* or *not performed at all*. The metrics associated with these three outcomes are *speed*, *reliability* and *availability*, respectively.

During our performance testing, we will only be concerned with the first kind of outcomes, thus we assume correct performance at all times. Performance analysis of correctly performed tasks may be measured by the time taken to perform the task (responsiveness), the rate at which the task is performed (productivity) and the resource consumption while the task is performed (utilization). These metrics are often referred to as *response time*, *throughput* and *utilization*, respectively. In this thesis, we will focus on the response time and the resource utilization achieved by our solution.

Metrics may be further divided into individual metrics, concerning the utilization experienced by one user only, or global, reflecting the utilization experienced by the system as a whole.

To get dependable results, experiments should be performed several times. The final result for one metric will then often be the mean value of the results from all experiments. However, one should be aware of the variance

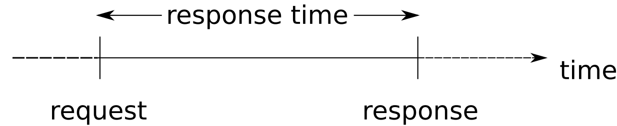


Figure 7.1: Response time metric.

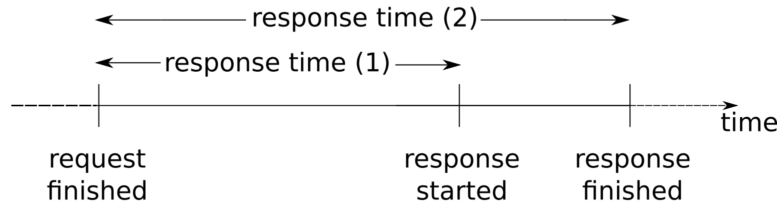


Figure 7.2: Response time ends when the response is started (1) and ended (2).

in the results, as outliers may affect the result significantly. During our experiments, all tests have been run 10 times in a row. All results are thus averages from 10 test runs.

7.6.1 Response Time

Response time is the interval between a user issues a request and a response is given by the system, as shown in Figure 7.1. This interval may be interpreted in two ways [17]: Either, it may be interpreted as the interval between the *end* of a request submission and the *beginning* of the corresponding response, or as the interval between the *end* of a request submission and the *end* of the corresponding response. The two different interpretations are shown in Figure 7.2.

Response time is often an individual metric, as it measures the time it takes for *one single* user to get a response, although one might also want to look at systemwide response time. In this thesis, we will only measure local response times.

We obviously want a resource localization to be performed as fast as possible, even though, as stated in the design requirements in Section 5.3, this is not our primary goal. There are two ways of measuring the speed of one localization:

- Actual time spent, in seconds. The results may not be too interesting as all testing is to be done with an emulator only. The time spent by the ant system relative to the time spent by the flooding system may, however, be interesting.

- Number of hops. This is straightforward in both systems - the number of hops used from a request is sent to an appropriate resource is

localized.

Measuring the Response Time

In the flooding approach, a resource reply is sent directly from the resource holder to the requesting node. In the ant solution, however, the backward ant travels the whole search path back to the requesting node (excluding any loops).

As mentioned above, there are two different definitions of response time, where the difference is where the response time measurement is stopped. When measuring the time spent on one resource localization, we thus need to define the term ”resource localization” - does this include the reply back to the requesting node? If the duration of the response is long, the preferred definition is the latter, where the duration of the system response is included in the response time. If we use this definition, the ant solution may need twice as many hops as if we excluded the reply. One might argue that these hops should be included, as it doesn't help the requesting node much that some node out there knows the location of the resource - it needs this information itself. However, there is an obvious possible improvement to the ant solution here: The resource holder may, in addition to sending the backward ant towards the source, also send a direct resource reply like in the flooding system. This introduces a slight overhead, as the requesting node will also receive the backward ant at a later point, but will improve the response time. This improvement is not implemented in our solution, but makes it possible to exclude the reply from the hop count and use the first definition - the response time is the interval from the forward ant is sent to it is received at a resource-holding node.

In the ant system, the requesting node sometimes needs to send more than one forward ant. During testing, we assume each forward ant that does not produce a reply has reached the time to live-limit, and thus count *time to live* hops times the number of non-returning ants plus the number of hops traveled by the ant that actually found the resource. Also, the response time in number of seconds is measured from the first forward ant is sent to search for a given resource.

Logging

Requesting nodes log the time at which a request is received, righ before the first forward ant or resource request is sent. Each ant and flooded request contains a field keeping track of the number of preceding nodes, and forward ants also how many ants have already tried to localize the resource without returning. Receiving resource holders log both the time of reception as well as the number of hops the request (counting ttl hops for every non-returning forward ant) has traveled. All values are written to the global result log.

7.6.2 Resource Usage and Utilization

The *utilization* of one resource is the fraction of time the resource is busy servicing requests [17]. As stated in the design requirements in Section 5.3, the main requirement for our solution is that a resource localization consumes as little resources, both local as well as network resources, as possible. Thus, we divide our resource usage measurements into two parts: Bandwidth usage and local processing power usage.

Bandwidth Usage

In general, the bandwidth usage should be measured from a search is started to the requesting node receives a reply. As opposed to the response time measurements, when measuring the resource *usage*, the reply traffic may not be omitted. Thus, for the ant system, the message count includes both forward as well as backward ants. If a search is initialized and no reply is received within *max_sleep_time* seconds, a new forward ant should be sent, and resource usage monitoring is continued until a reply is received. It might happen, however, that a new search is started before a backward ant from a previous search reaches the requesting node. Thus, we need to keep monitoring until there is no network activity.

The same holds for the flooding solution. The message count consists of the number of broadcast messages plus the one message needed to reply to the requesting node. We cannot stop the monitoring when a reply is received, as the flooding might still be going on in other parts of the network — other nodes will not know that a resource is already localized. Thus, as in the ant solution, we need to keep monitoring until there is no activity in the network.

Message Count To get a view of the bandwidth usage of the two systems, the number of messages *sent* in the entire network during one resource localization is measured. By measuring the number of *sent* messages, a broadcast message will count as *one* message, no matter how many nodes receive the message.

The message count MC_{bw} in a network of n nodes is thus:

$$MC_{bw} = \sum_{i=1}^{n} (messages\ sent\ from\ n). \tag{7.1}$$

Byte Count Counting the number of messages may give a good picture of the resource usage of the two systems, but the resource usage depends not only of the number of messages sent, but also the size of the messages. Sending a few large messages might take just as much resources as sending several smaller messages. Thus, we also measure the number of bytes sent

during resource localization. Comparing the number of bytes sent with the number of messages sent tells us if the message count is accurate enough, or if the size difference is large enough to affect the results of the two systems.

The byte count for bandwidth usage, BC_{bw} in a network of n nodes is thus:

$$BC_{bw} = \sum_{i=1}^{n} (bytes\ sent\ from\ n). \tag{7.2}$$

Utilization In general, the message count alone is not enough to conclude on the bandwidth utilization. We need to know how many of the messages that were sent that were actually used for something.

In the ant system, all messages are either forwarded or replied to, except for forward ants with $ttl = 0$. Thus, we expect almost full bandwidth utilization. In the flooding system, however, a sent message is regarded utilized, in bandwidth manner, if at least one of the recipients either re-broadcast it or answers to it. The utilization U_{bw} in a network with n nodes is thus:

$$U_{bw} = \frac{\sum_{i=1}^{n} (messages\ replied\ to\ at\ n)}{\sum_{i=1}^{n} (messages\ sent\ from\ n)}. \tag{7.3}$$

However, the ant solution bandwidth utilization should be ≈ 1, as all received messages should be either answered to, if the resource is present, or forwarded, if the resource is not present. The only reason for this not to happen is if the forward ant has a ttl = 0. In this case, the ant dies and the whole search is wasted, not only the last jump. Thus, we have not conducted any measurements of the bandwidth utilization.

Local Processing Power Usage

The measured bandwidth usage provides information on the network load. However, one characteristic of wireless transmission is that all nodes within range of a transmitter will pick up the message and check if it is destined for itself. If it is, the resources used to handle this message were not wasted. However, if the message was not destined for this node and is thus thrown away, this resource usage was a complete waste.

This characteristic is not very well exploited in the ant solution. Here, all messages are destined for one specific node. Thus, there is one recipient, so the message will not be wasted in terms of bandwidth, but all nodes that pick up the message but find it to be destined for some other node, do waste some processing power on this message handling.

In the flooding solution, on the other hand, this characteristic is exploited and used to spread the request as fast throughout the network as possible. Still, a lot of messages that have already been re-broadcast may be received again from other sources, and are thus wasted, as each node will only re-broadcast the same message once.

There is a difference between wasted messages in the ant solution and wasted messages in the flooding solution, however. In the ant solution, wasted messages are thrown away at the network layer when the routing protocol discovers that this message was not destined for this node. In the flooding solution, however, ”waste” is not discovered before the message is to the application layer and the application finds out this message has already been re-broadcast, and thus waste even more resources than those in the ant application. However, for simplicity, in our analysis, all wasted messages are regarded equal.

Message Count In this case, we count the number of *received* messages. Even if a message is broadcast and is thus only sent once from the source, every receiving node needs to handle the message.

The message count MC_{lpp} in a network if n nodes is thus:

$$MC_{lpp} = \sum_{i=1}^{n}(messages\ received\ at\ n). \tag{7.4}$$

Byte Count As in the bandwidth measurements, we also look at the number of bytes sent by the two systems. The byte count for local processing power usage, BC_{lpp} in a network of n nodes is:

$$BC_{lpp} = \sum_{i=1}^{n}(bytes\ received\ at\ n). \tag{7.5}$$

Utilization As with bandwidth, we want to know how good the local processing power *utilization* is. The utilization is the fraction of messages or bytes that are actually ”used” to the total number of received messages or bytes. Thus, the utilization U_{lpp} in a network of n nodes is:

$$U_{lpp} = \frac{\sum_{i=1}^{n}(messages\ replied\ to\ at\ n)}{\sum_{i=1}^{n}(messages\ received\ at\ n)}, \tag{7.6}$$

or alternatively

$$U_{lpp} = \frac{\sum_{i=1}^{n}(bytes\ replied\ to\ at\ n)}{\sum_{i=1}^{n}(bytes\ received\ at\ n)}. \tag{7.7}$$

Topology Update Messages

In the ant system, we communicate with olsr to get topology information. As this is only communication between two ports on the same (virtual) node, it only imposes local load on each node, not on the network, and these messages are thus omitted from message and byte counts.

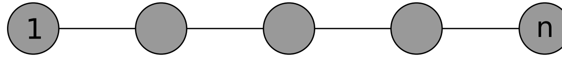


Figure 7.3: Chain topology.

7.7 Test Scenarios

The test scenarios are the virtual environments in which the solutions are tested. The scenarios define the network topology and node mobility, as well as the node behavior, meaning resource localization initialization.

For the tests to give as realistic results as possible, the node mobility and behavior should be as close to that in real life usage of the system. The best is to use real world traces to define node mobility. However, real world traces are not highly available, and are typically quite advanced. In this thesis we instead use only a few, quite basic scenarios to show and verify basic functionality and the most important principles. One scenario includes node mobility, but no node joins or leaves are included in any scenario.

7.7.1 Chain Scenario

The simplest possible topology is a static chain, as shown in Figure 7.3. In this scenario, every node has minimum one neighbor (the chain ends) and maximum two neighbors (the middle nodes).

In this scenario, we expect both solutions to behave similarly until a resource is located, as the only possible path for the forward ant to travel is straight through the chain, which should also be the path of the broadcast messages.

7.7.2 Grid Scenario

In the grid scenario, in which we still use static nodes, the number of neighbors per node increases. The four corner nodes have two neighbors, other edge nodes have three neighbors and the middle nodes have four neighbors, as shown in Figure 7.4. The number of possible paths for an ant increases drastically, and so does also the number of re-broadcasting nodes in the flooding solution.

In this setup, we expect the forward ants to use a longer time to find a resource, as it needs a bit more luck to find what it is looking for. Depending on the *time to live* value, we might also need to issue several forward ants in order to find the requested resource. However, if several ants are looking for the same resource, we should see a decrease in the response time over time, as resource location information is spread throughout the network.

The flooding solution, on the other hand, is expected to give relatively quick answers, but also to put a lot more load on the network. The response

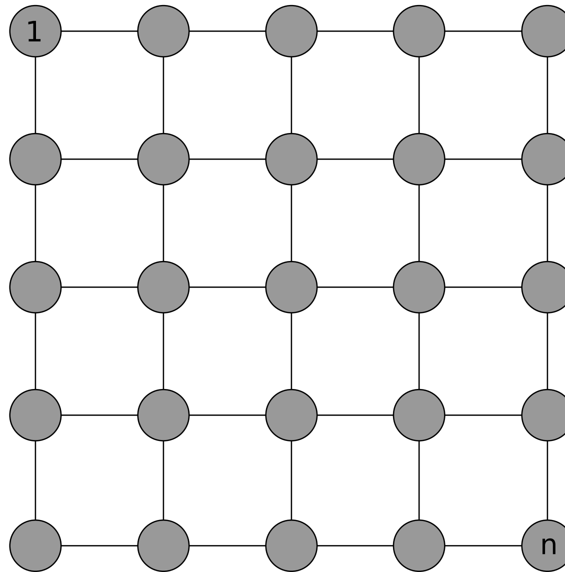


Figure 7.4: Grid topology.

time should also be constant, as no nodes learn anything from previous resource localizations.

7.7.3 Mobility Scenario

In this scenario we introduce mobility. All nodes are moving according to the random waypoint model [18].

What we want to do in this scenario, is to exploit the movement of the middle nodes and the fact that they, when using the ant solution, will remember the location of the resources that have been located in the near past. To do this, we use several requesting nodes, initially located a bit apart from each other in the network (but still in the same partition). The resource is placed quite close to the first requesting node, but still a few hops away to make sure we actually have some nodes containing the resource information after the first search. An example initial setup is shown in Figure 7.5.

What we expect in this scenario is that the average response time and resource usage decreases with the number of requesting nodes, as these utilize stored resource information.

The flooding solution will not be able to exploit the mobility of the nodes in any way, and we thus expect average response time and resource usage to be more or less constant even if the number of requests increases.

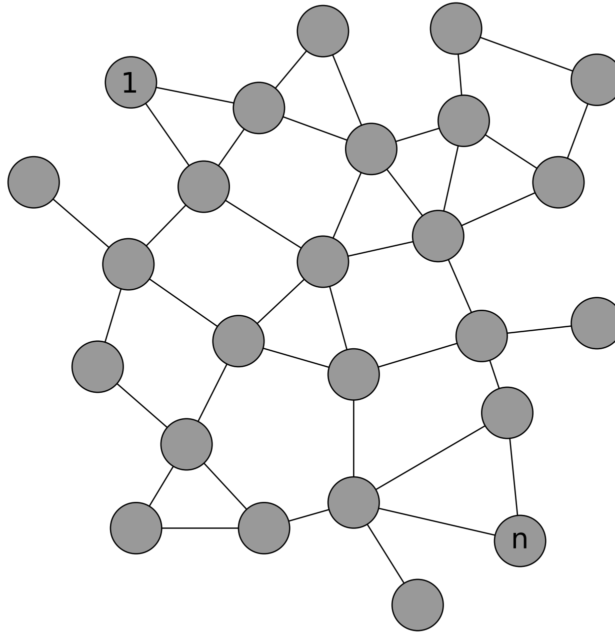


Figure 7.5: Mobility topology.

7.8 Test Scenario Implementation

As stated above, we have used *setdest* to create our scenario files. To construct the chain and grid scenarios, however, changes have been made by hand to achieve the desired topology. In each scenario, the resource-requesting nodes issue their requests one after one, with a given time interval in between.

As described in Chapter 6, a resource localization is initialized by sending a UDP segment to port 3457 on the resource-requesting node. In our tests, this is done by including the following line in the scenario file:

```
$ns_ at X "$node_(Y.Y.Y.Y) sendmsg port=Z msg='name quality'"
```

where X is the number of seconds into the emulation where the message should be sent, $Y.Y.Y.Y$ is the IP address of the node the message should be sent to, Z is the port number of this host and *name* and *quality* identifies the resource to be searched for.

For example, the following line:

```
$ns_ at 1.0 "$node_(10.0.0.1) sendmsg port=3457 msg='cpu 10'"
```

sends the message "cpu 10" to the node with IP address 10.0.0.1 on port 3457 one second into the emulation, telling the node to initialize a search for resource cpu with a minimum quality of 10.

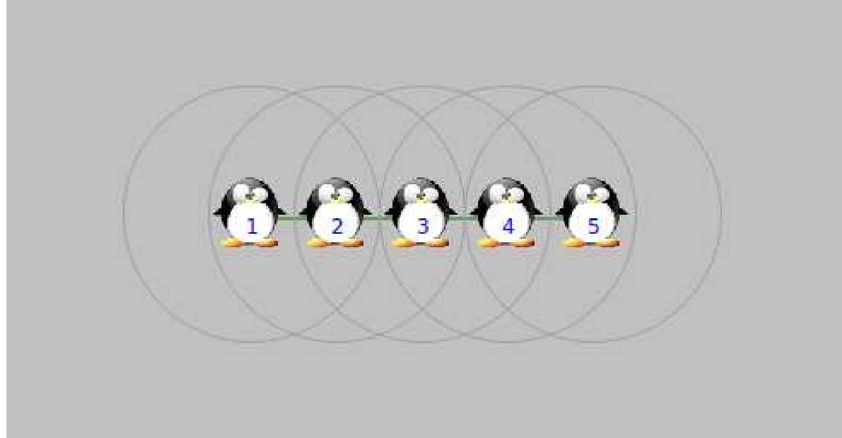


Figure 7.6: A screenshot from the NEMAN gui showing the chain scenario.

The actual scenario file for the chain scenario is listed in Section A.1. Because scenarios tend to grow very large when introducing more nodes and especially mobility, the scenario files for the grid and mobility scenarios are not listed. However, the main difference is that these two files include more link "definitions" and, in the case of mobility, link and range updates at certain points in time.

7.8.1 Static Scenarios

A screenshot from the chain scenario as presented by the NEMAN gui is shown in Figure 7.6. Each node is represented as a penguin, the circles show the range of each node and the green lines show which links are present between the nodes in the scenario.

A similar screenshot of the grid scenario is shown in Figure 7.7.

7.8.2 Mobility Scenario

In NEMAN scenario files, there are two kinds of ranges: *data-range* and *broadcast-range*. For the ant solution and the flooding solution to be comparable, these two ranges need to be the same, as the former uses regular data packets and the latter uses broadcasting. In a scenario file generated by setdest, a range of 1 between two nodes means that they are within data-range, whereas a range of 2 means they are only within broadcast-range. To solve this issue, we use a script to simply remove all entries from the scenario file where the range is set to 2. All higher ranges mean that the two nodes are completely outside the range of each other, and are thus left as they are.

A NEMAN gui-screenshot of the mobility scenario is shown in Figure 7.8. Figure 7.8a shows the initial state of the network, whereas Figure 7.8b shows the final network state, after 120 seconds of emulation. Note that the

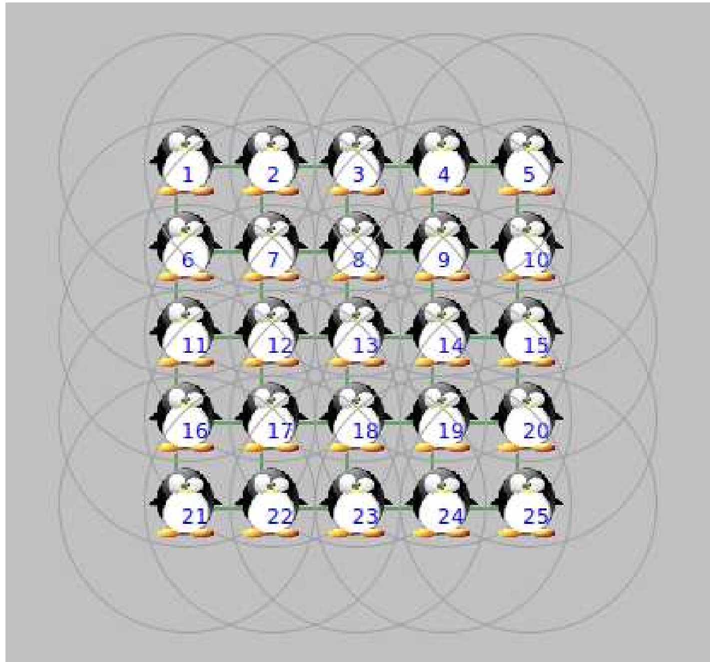


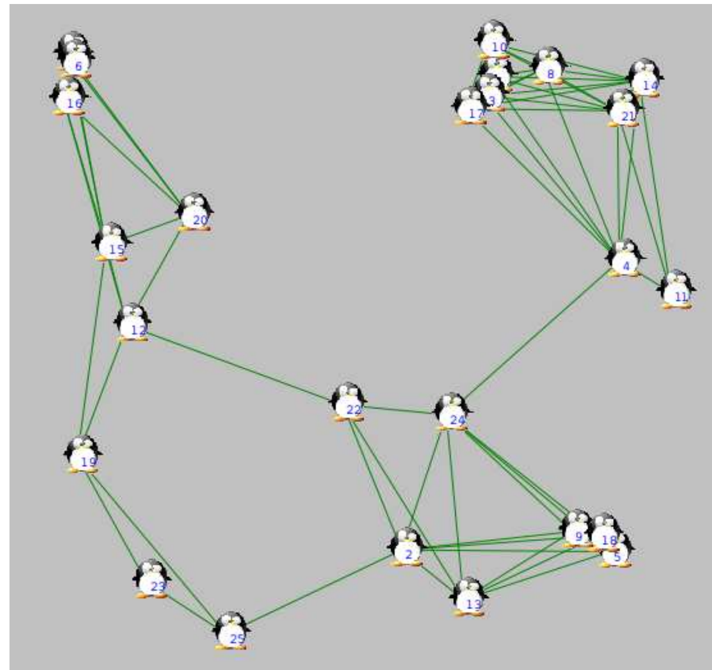
Figure 7.7: A screenshot from the NEMAN gui showing the grid scenario.

ranges of the nodes are left out, as for some reason, the ranges generated by setdest do not always correspond with the physical node locations, also provided by setdest — sometimes we get a link between nodes that should have been outside the range of each other.

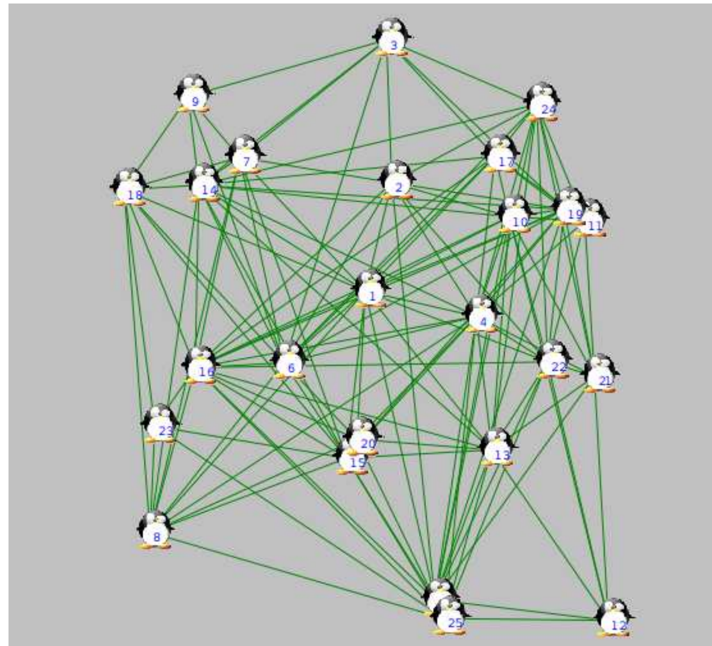
7.9 Workload

The *workload* is the load that is put on the system during usage. We may differ between two kinds of workload: *test workload* and *real workload* [17]. The real workload is the workload that the system will be exposed to during normal operation, whereas the test workload is the workload under which the system is tested, and which is used to compare different systems. For the tests to give as realistic results as possible, the test workload should be similar to the real workload. The test workload may, however, just like test scenarios, be simplified to ease the configuration of the workload.

In our experiments, the workload is made up by the number of resource-holding nodes, resource-requesting nodes, number of requests and node location and mobility. As stated in the design assumptions in Section 5.2, we assume that both resource-requesting and resource-holding nodes are randomly placed within the MANET. To reflect this, we should place these at random positions in our scenarios. However, because we need our tests to



(a) Initial state.



(b) Final state.

Figure 7.8: Screenshots from the NEMAN gui showing the mobility scenario.

be repeatable, random placement is not desireable. Instead, we have tried to place them as ”randomly” as possible. This will be further explained for each scenario below.

In all scenarios, we have, for simplicity, restricted ourselves to only one resource-holding node per MANET. The number of resource-requesting nodes varies, as explained below. Tests are divided between *single location*, with only one resource-requesting node, and *location learning*, where there are several resource-requesting nodes. In all experiments, the requesting node is looking for a resource named ”memory” with a quality of at least 90. The resource-holding node holds a resource with name ”memory” and a quality 100 + its tap number, thus, resource memory at tap number 4 has a quality of 104. The only reason for this is to make it easier to verify when reading output files that the resource was located at the correct node — the only actual requirement is that the quality is ≥ 90.

7.9.1 Parameters

For all tests, we have set the time to live value for all ants and flooded messages equal to the number of nodes in the network minus one. If this value is too small, the possibility that an ant actually finds a resource is small, as it might never get to walk far enough. If the value is too large, we risk that ants ”get lost” in parts of the network where the resource is not present. We have chosen to set this value equal to the number of nodes in the MANET. The ant should then have the possibility to walk the entire diameter of the network, but still not walk ”forever”.

The expiry time of learned resource location information is set to infinity. This is a rather unrealistic value, but makes our test cases simpler.

The max_sleep_time for the ant solution is set to 2 seconds.

7.9.2 Scenario Properties

Chain Scenario

As this is a static scenario, the size of the area in which the network is located is unimportant. The scenario consists of five nodes linked together in a chain. The purpose of this setup is mainly to show that the algorithm is sound and the functionality works as expected. Because of the characteristics of the chain topology, we do not conduct any experiments with more than one requesting node.

Grid Scenario

Our grid scenario consists of 25 nodes, in a 5 x 5 grid.

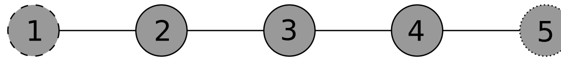


Figure 7.9: A chain topology with requesting and resource-holding nodes at the ends. A dashed line represents a requester, and a dotted line represents a resource holder.

Mobility Scenario

This scenario consists of 25 mobile nodes. These move within an area of size 700m x 700m, with a maximum speed of 15 metres per second and a maximum pause time of 8 seconds for a duration of 120 seconds.

7.9.3 Single Localization

In these experiments, we use one resource-requesting and one resource-holding node for all three scenarios.

Chain Scenario

In our simplest experiments, we use a chain with five nodes, where we place the requesting node at one end of the chain and the requested node in the other end, as shown in Figure 7.9.

Grid Scenario

There are several possibilities when it comes to placing the requesting node as well as the resources requested in the grid. With only one available resource, one possibility is to place the requesting node and the requested resource in diagonally opposite corners. To obtain a more ”random” placement of these two nodes, we have placed them one hop into the network, giving them four instead of two neighbors, as shown in Figure 7.10, which is more likely than a real corner placement with only two neighbors. Thus, in the figure, node 7 is the requesting node and node 19 is the resource-holding node.

Mobility Scenario

In an experiment with one single localization, we do not really experience any effects of the mobility. We have still included this experiment for completeness' sake. The scenario also provides a picture of how the ant solution behaves in a network with a more irregular topology, where some nodes have few neighbors and others may have more neighbors.

Resource-requesting and resource-holding nodes have been chosen such that intially the shortest path between them is 4 hops, as this gives a large number of possible paths for the forward ant. Thus, this case is quite similar to the grid scenario when performing only one localization. The position of

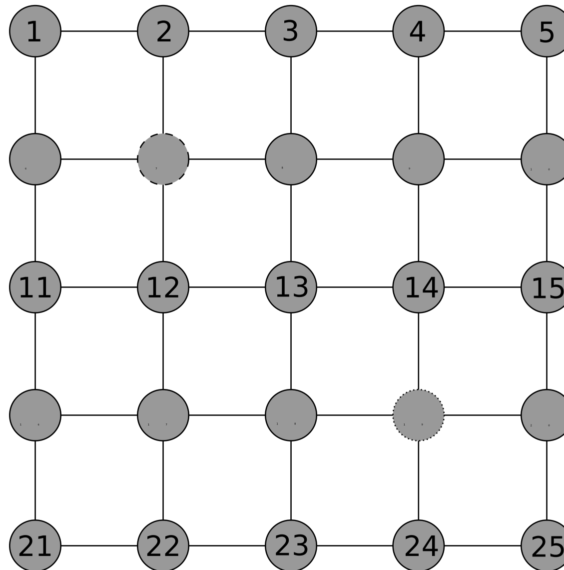


Figure 7.10: A grid topology with 25 nodes: One requesting node (node 7) and one resource-holding node at diagonally opposite "corners". A dashed line represents a requester, and a dotted line represents a resource holder.

these nodes are shown in Figure 7.11, where node 12 is the requesting node and node 4 is the resource-holding node.

7.9.4 Location Learning

In this range of test scenarios, we introduce several resource-requesting nodes. These all request exactly the same resource. Thus, in the ant solution, as soon as the first forward ant has located the resource, the location information starts spreading throughout the network.

Grid Scenario

To show the real effect of the ant approach and resource location learning, we use a grid with three different requesting nodes. These are placed in near-"corners" (nodes 7, 9, and 17) except for the one holding the resource-holding node (node 19), as shown in figure 7.12. The shortest path from these nodes to the resource-holding node is thus 4, 2 and 2 hops, respectively.

The requesting nodes issue their requests in numerical order, starting with the lowest numbered node. Requests are issued every 15 seconds, giving each node up to 8 tries to locate the given resource before the next node starts. If the first node does not locate the resource in 8 tries, the requesting nodes will continue resource localization simultaneously.

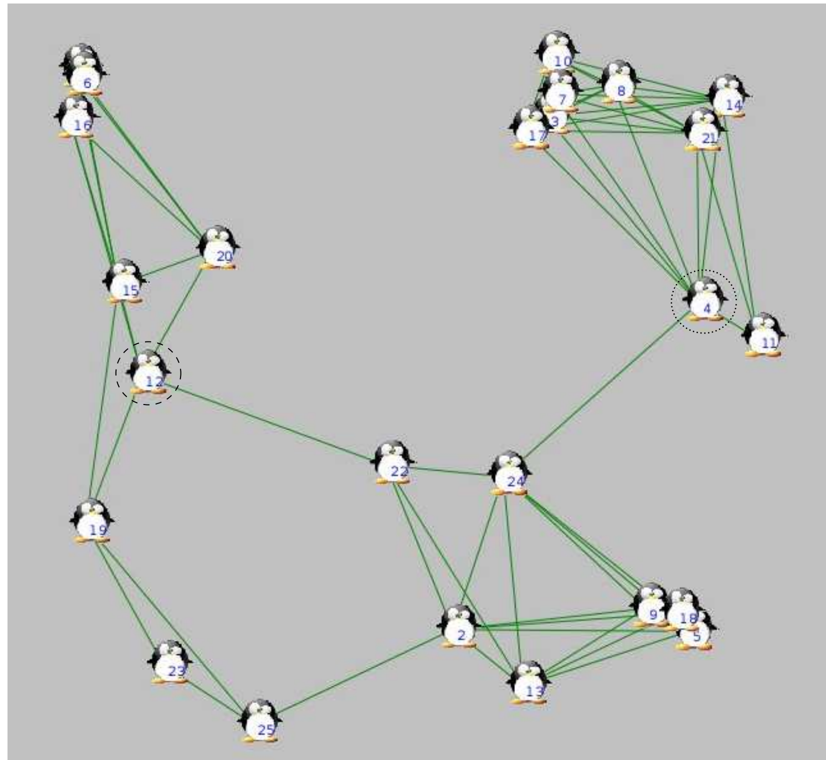


Figure 7.11: A mobility topology with 25 nodes; One resource-requesting node (node 12) and one resource-holding node (node 4). A dashed circle represents a requester, and a dotted circle represents a resource holder.

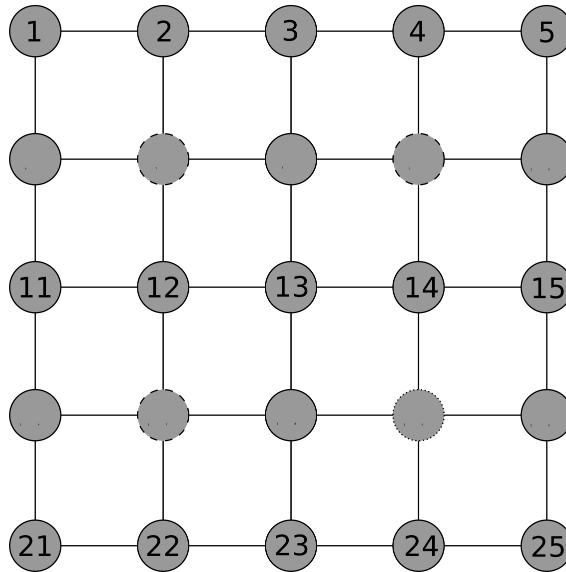


Figure 7.12: A grid topology with 25 nodes; Three resource-requesting nodes (nodes 7, 9 and 17). one resource-holding node (node 19). A dashed line represents a requester, and a dotted line represents a resource holder.

Mobility Scenario

Initially, the resource-requesting nodes are placed around the edge, and the resource-holding node in the ”middle”. We use five different resource-requesting nodes, each requesting a resource with 25 seconds in between. This gives nodes time to move a bit around between each search, and thus spreading any learned location information. However, even if requesting nodes were initially placed around the edge, this may have changed before they actually get to issue a request. The scenario with requesting and resource-holding nodes in their initial positions is shown in Figure 7.13.

Like in the grid scenario, nodes receive the initial resource requests in numerical order, starting with the lowest numbered node. To give nodes time to move around a bit between each initiated resource localization, requests are issued by different requesting nodes every 25 seconds.

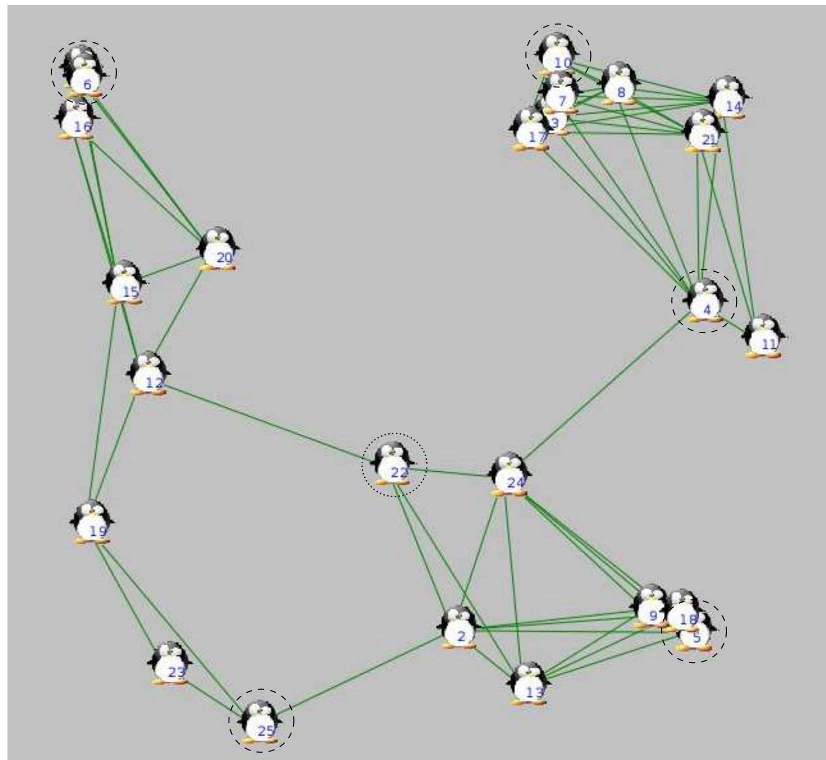


Figure 7.13: A mobility topology with 25 nodes; One resource-holding node (node 22) and five resource-requesting nodes (nodes 4, 5, 6, 10 and 25). A dashed circle represents a requester, and a dotted circle represents a resource holder.

Chapter 8

Performance Evaluation

> *"Software and cathedrals are much the same — first we build them, then we pray."*
>
> — *Sam Redwine*

In this chapter we present the results from our performance tests. We start by explaining some factors that may have an impact on the tests results. Then we compare the results from the ant and flooding solutions, and look at which solution performs best in each test scenario and analyze why the solutions perform as they do. We also give a few suggestions to what may be done differently to address some of the problems observed when analyzing the test results.

8.1 Influencing Factors

8.1.1 Topology Initialization and Updates

During our testing, we have experienced some problems with OLSR using a very long time to get all topology information and set up routes. In the very simple chain scenario with only five nodes, it took up to 15 seconds to establish a route between nodes 1 and 5. For the ant solution, this is not a problem, as ants move between neighbors, and neighbor information appears much faster than longer distance routes. However, for the flooding solution's resource replies to get delivered from the resource-holding to the resource-requesting nodes, we need several hops-routes to be established.

To solve this, we have introduced a delay between the time a scenario is started and the first resource request is sent from the scenario file to a resource-requesting node. In the stationary scenarios this is trivial, as the network looks the same after 30 seconds as it did when the scenario was started. In the mobility scenario, however, the node positions may be very

different after 30 seconds. In the single localization mobility scenario, this is not a problem, as we still do not exploit the mobility. Note, however, that the topology when the resource request is sent is *not* the same as the initial topology shown in Figure 7.11.

In the location learning mobility scenario, however, we "loose" 30 seconds of mobility, leaving less time to allow nodes to move in between resource localizations. To avoid this, we have added 30 seconds of non-mobility to this scenario file between the node range initialization and the first range updates. Thus, this scenario actually lasts for 150 seconds instead of 120, but only the last 120 seconds are exploited by the ant and flooding solutions.

In Section 7.6.1, we stated that during performance testing, we will only be concerned with successful tests. In Section 6.6.3, we stated that node mobility should not be a problem for the ant solution, as any lost neighbors would be handled by the routing protocol. It turns out, however, that at some occasions, topology updates are too slow, and some ants are thus lost when a node tries to forward it to a newly lost neighbor. Unfortunately, time issues inhibited us in adjusting the code to this. During our testing, this happened in around 2 out of 10 single test runs. The experiments where this occurred do not count as "successful": If an ant is lost, a new forward ant will be sent by the requesting node, so a resource will still eventually be found if present. However, the system will count the lost ant as an ant reaching time to live = 0 and assume it has traveled its maximum number of hops, which may not be correct. To get as correct performance results as possible, we have thus left out those experiments where an ant is lost. We have thus run more than 10 experiments in these cases, and left out those where an ant was lost due to transmission errors. Making the ant solution more robust to such transmission errors is thus left as further work.

8.1.2 Time Inaccuracy

The timing of one resource localization has been performed by using the **gettimeofday()** function. This is possible without any clock synchronization as the whole system is running on the same machine. However, due to differences in the ant and flooding solution source code, it is not possible to start and stop these measurements at the exact same position in both systems. Thus, these are approximate times.

8.1.3 Average Neighborhood Size

For the chain and grid scenarios, the number of neighbors per node is constant, as there is no mobility or link changes. In the chain scenario, the average number of neighbors per node is 1.6, and in the grid scenario the average number of neighbors per node is the double, namely 3.2. In the mobility scenarios, however, the number of neighbors per node varies over

time.

As our ant solution is targeted at MANETs consisting of only one partition, we need our mobility scenario to stay completely connected during the entire emulation. This does not correspond with the *random movement* of the nodes, and as far as we are concerned, there is no easy way to automatically generate a scenario with mobility and with the guarantee that nodes always stay within the same partition. Thus, when making the mobility scenario, we had to set the area small enough that partitioning was highly unlikely. The result was a scenario file with only one partition, but with a very high connectivity, meaning a very high number of neighbors per node. Later in this chapter we will see that this has a negative impact on the ant solution. Basically, this is because a high number of neighbors decreases the possibility that an ant choosing a next hop neighbor to move to actually makes a "clever" choice, meaning a next hop that moves the ant closer to the requested resource.

Because of these problems, we have made an alternative mobility scenario. Initially, this scenario was a copy of the original scenario. However, we have made a simple script that loops through all lines in the scenario file, and whenever a line setting the range between two nodes to one is discovered, this line is removed from the file with a probability of 0.7. Thus, a lot of links from the original scenario are removed, making the scenario less connected. For the rest of this thesis, this new scenario will be denoted a "sparse mobility scenario", whereas the original scenario is denoted a "dense mobility scenario". Figure 8.1 shows the average number of neighbors per node per second into the emulation for the two mobility scenarios. Note that the data from the scenarios have been discretized by rounding all timestamps down to the nearest whole second. In the dense scenario, we see that the number of neighbors increases from around 4 to between 12 and 14, which is a very high number in a network with only 25 nodes. In the sparse scenario, the number of neighbors is relatively constant.

For all location learning scenarios, we have run all tests both for the sparse and the dense mobility scenario. For the single localization scenarios, however, we have only used the dense mobility scenario.

In the mobility scenarios with location learning, five different nodes request a resource with 25 seconds between. Table 8.1 shows how many neighbors nodes have, on average, when each of the nodes initialize its search.

8.2 Results — Response Time

8.2.1 Single Localization

In the single localization experiments, all program instances were completely restarted for each experiment, so no resource information is learned between two resource localizations. The average number of hops per resource local-

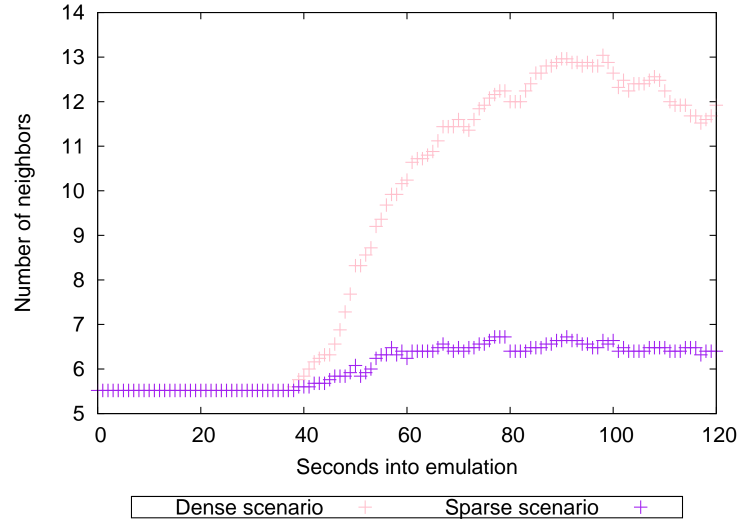


Figure 8.1: Average number of neighbors per second into the simulation in the mobility scenarios.

Request number	1	2	3	4	5
Dense scenario	5.52	9.36	12.0	12.4	12.48
Sparse scenario	5.52	6.32	6.4	6.4	6.4

Table 8.1: Average number of neighbors per node in the dense and sparse mobility scenarios when requests are issued.

Solution	Chain Scenario	Grid Scenario	Mobility
ant	4.0	21.0	51.8
flood	4.0	4.0	2.0

Table 8.2: Average number of hops used in one resource localization in each scenario.

Solution	Chain Scenario	Grid Scenario	Mobility
ant	13.02	848.8	3130.0
flood	1.5	14.3	13.2

Table 8.3: Average time in milliseconds spent on one resource localization in each scenario.

ization for each scenario is shown in Table 8.2, and the average time spent per resource localization for each scenario is shown in Table 8.3.

Chain Scenario

As we expected, the two solutions use the same number of hops in the very simple chain scenario. This scenario consists of five nodes that all need to be visited to find the resource in the other end, giving four hops. This fits perfectly with the observed results.

For the time spent, however, we see that the flooding solution uses less time than the ant solution: The ant solution uses almost nine times more time than the flooding solution. This is probably due to the fact that the ant solution requires quite a lot more processing in each intermediate node than the flooding solution, which only re-broadcasts received messages. The ant solution also handles incoming topology updates, and thus simply has more to do than the flooding solution.

Grid Scenario

For the grid scenario, the difference in the response time for the two solutions really starts to show. The response time in number of hops is 7.7 times as large in the ant solution as in the flooding solution. As expected, the flooding solution still uses 4.0 hops, which is the length of the shortest path from the resource-requesting to the resource-holding node, in average to localize a resource. The ant solution uses almost 60 times as much clock time to localize a resource as the flooding solution.

Mobility Scenario

In these scenarios, the difference between the response time in the ant solution and the flooding solution gets even more significant. The flooding solution, which always reaches the resource-holding node in the shortest possible number of hops, has a response time of 2 hops in the dense scenario, whereas the ant solution uses on average 51.8 hops, or almost 26 times more hops than the flooding solution. In the sparse scenario, the flooding solution uses 3.2 hops, whereas the ant solution uses 105.67 hops, approximately 33 times as many as the flooding solution. The same is shown by the measured times: In the sparse scenario, the ant solution uses a whooping 381.69 times longer to locate the given resource.

This is a scenario with mobility. However, as we are only performing one resource localization, the duration of an experiment is too short for the mobility to make a difference on the results. However, there is a possibility that the topology changes between two subsequent forward ants if more than one is needed. The network topology might thus be different when the last forward ant is sent than it was during the resource localization in the flooding solution, making the minimum number of hops smaller or larger.

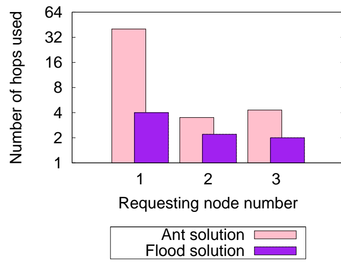
8.2.2 Location Learning

Grid Scenario

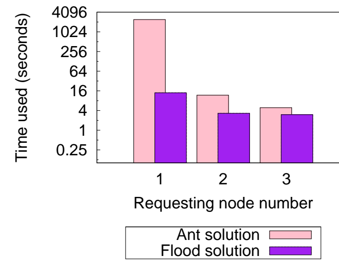
Figure 8.2 shows the average response time against the number of requesting nodes in the grid scenario with location learning. The response time as number of hops is shown in Figure 8.2a, and the response time in milliseconds is shown in Figure 8.2b. Response times are plotted *per resource-requesting node*, thus the first pair of columns shows average response time for the first requesting node, the second pair of columns shows the average response time for the second requesting node and so on. Note also that the response times in both figures are plotted with base 2 logarithmic scale. This is done because of the large span in response time results: When using a linear scale, most results do not show on the plot because of the large difference in response time from the first resource-requesting node in the ant solution to all subsequent resource-requesting nodes in the ant solution and all flooding solution results.

From the plot, we see that for the ant solution, the tendency in this scenario is quite clear: The response time, both in number of hops as well as in milliseconds, decreases drastically from the first, with 40.2 hops, to the second requesting node, with 3.5 hops. From the second to the third requesting node, there is little difference, and we actually see a small increase in number of hops, but a small decrease in number of seconds.

For the flooding solution, however, we get the expected results: The first requesting node uses more hops, but this is due to the fact that the

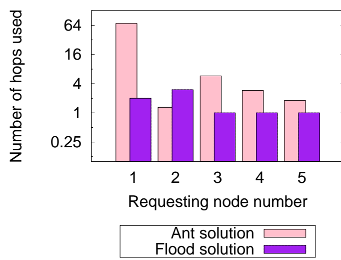


(a) Number of hops.

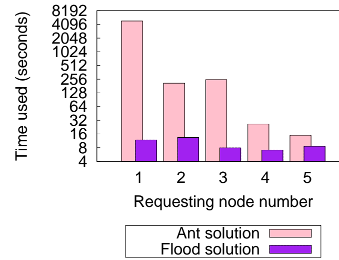


(b) Number of milliseconds.

Figure 8.2: Average response time per resource-requesting node in grid scenario with location learning. Note: y-axis plotted with base 2 logarithmic scale.



(a) Number of hops.



(b) Number of milliseconds.

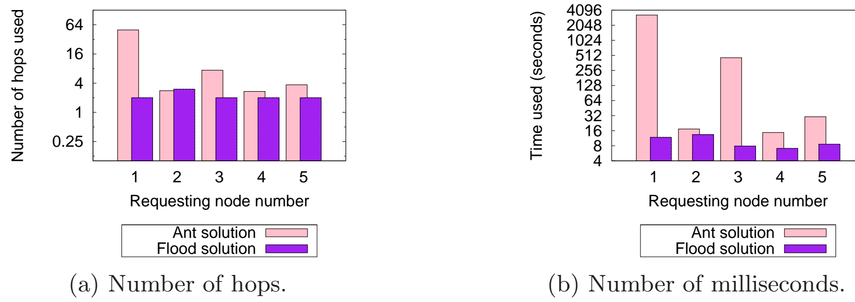
Figure 8.3: Average response time per resource-requesting node in dense mobility scenario with location learning. Note: y-axis plotted with base 2 logarithmic scale.

first requesting node is located 4 hops away from the resource-holding node, whereas the second and third are only two hops away. In one of the flood experiments, one flooded package from the second requesting node reached the resource-holding node via a longer path before the one traveling the shortest path. The flooding solution only handles the first incoming copy of the same message, thus, there is a slight difference between the average number of hops for the second (2.2 hops) and the third (2 hops) requesting node.

Mobility Scenario — Dense

In Figure 8.3, we see how the average response time per resource localization changes with the number of nodes requesting the same resource. Figure 8.3a shows the response time in number of hops, whereas Figure 8.3b shows the number of milliseconds spent.

We see here the same tendency as we saw in the grid scenario: There is a dramatic decrease in the response time from the first to the second



(a) Number of hops. (b) Number of milliseconds.

Figure 8.4: Average response time per resource-requesting node in sparse mobility scenario with location learning. Note: y-axis plotted with base 2 logarithmic scale.

requesting node. However, here we see that the response time continues to decrease when introducing more requesting nodes.

In this scenario, we also see a slight decrease in the response time for the flooding solution. The differences here are, however, a lot smaller, and appear mainly because the scenario goes from being quite ”stretched” in the beginning, to being more tightly connected towards the end. Thus, the minimum distance between two nodes decreases slightly over time.

Mobility Scenario — Sparse

Figure 8.4 shows the average response time per resource localization per resource-requesting node in the sparse mobility scenario. Figure 8.4a shows response time in number of hops, whereas Figure 8.4b shows the response time in milliseconds.

Like in the dense mobility scenario, the response time for the flooding solution is relatively constant. For the ant solution, however, we see the same drastic decrease in response time from the first to the subsequent requesting nodes. Measured in milliseconds we see a peak for requesting node number three, but this is due to the *max_sleep_time*, which is relatively long compared to the time it takes for a forward ant to walk its maximum number of hops. Measured in number of hops, we only see a slight increase for the same requesting node.

8.2.3 Conclusion

For both scenarios we see that the flooding solution always gives a shorter response time than the ant solution. However, the difference between the two decreases with the number of requesting nodes.

On average, one localization in the mobility scenario using the ant solution takes 3.13 seconds. In a scenario of this size, the ant solution is likely

to need more than one forward ant to locate a resource because an ant's time to live-value reaches 0. The response time in milliseconds is thus highly dependent on the size of max_sleep_time, the time before a forward ant is considered unsuccessful by the requesting node and a new forward ant is sent. As the max_sleep_time between two forward ants in our tests is 2 seconds, this tells us that in this scenario, the ant solution on average needed more than one forward ant to locate the resource (it is highly unlikely that the first ant used such a large amount of time to move the maximum of 25 hops). Thus, had the max_sleep_time been different, then the response time would be equally different.

Another observation is that the ant solution performs particularly bad in the mobility scenarios, especially in the single location case but also in the location learning case, there especially for the first requesting node. In Section 8.1.3, we saw that the number of neighbors per node is a lot higher in the dense mobility scenario than in the other two scenarios, and also a bit higher in the sparse mobility scenario. For a forward ant, this means that when standing in a node selecting which neighbor to use as next hop, it has a high number of neighbors to choose from. With only one resource-holding node present in the entire network, the chance of making a clever choice in this situation decreases with the number of neighbors. From the tcpdump files from the mobility scenario, we see that in many cases, the first requesting node may have to send up to six forward ants to look for the same resource before it is actually located. The reason for this is that there are too many available neighbors and thus too little chance of making the "right" choice — the forward ant needs a bit of luck to be able to locate the resource fast. The node needs more tries to find out which paths lead to the resource and which paths do not.

This is also reflected by the difference between the two mobility scenarios: For the first requesting node, the number of hops is 69.7 and 50.1 for the dense and sparse scenarios respectively. There is thus a decrease in the number of hops needed in the sparse scenario. The much smaller number of neighbors in the sparse scenario increases the possibility that a forward ant makes a "smart" choice when choosing a next hop. As soon as the location learning kicks in, there is little difference between the sparse and the dense scenario.

8.3 Results — Bandwidth Usage

8.3.1 Single Localization

The results from our bandwidth experiments are shown in Tables 8.4 and 8.5, under "Messages Sent" and "Bytes Sent". These show the total number of messages and bytes that were, in some way, sent from all nodes in the scenario during the different scenarios. By "sent", we mean both from our

Scenario	Solution	Messages Sent	Messages Received	Utilization
Chain	Ant	8.0	14.0	57.14
	Flood	8.0	14.0	57.14
Grid	Ant	26.7	91.7	29.12
	Flood	35.0	92.0	38.04
Mobility	Ant	56.6	660.1	8.57
	Flood	40.0	271.0	14.76

Table 8.4: Messages sent and received and message utilization for both solutions in each single localization scenario.

Scenario	Solution	Bytes Sent	Bytes Received	Utilization
Chain	Ant	1216.0	2128.0	57.14
	Flood	524.0	1400.0	57.14
Grid	Ant	6584.0	22440.0	29.34
	Flood	3604.0	9680.0	37.23
Mobility	Ant	15940.8	186308.8	8.56
	Flood	4064.0	28852.0	14.09

Table 8.5: Bytes sent and received and message utilization for both solutions in each single localization scenario.

application as well as those messages that were only forwarded to other nodes on the transport layer.

Note that as a neighbor may be lost between the forwarding of a forward ant and the reception of the corresponding ant, as explained in Section 6.6.3, the number of sent messages may be slightly larger than the actual number of nodes visited by the forward ant, as the backward ant may need to be forwarded via other nodes to reach the lost neighbor.

If we look at the number of bytes sent, we see that in each scenario, the flooding solution performs significantly better than the ant solution, even though the ant solution nodes sent less messages in the grid scenario. This is due to the large message size in the ant solution: A forward ant is initially 128 bytes large, whereas a flood solution resource request is only 108 bytes. In addition to this, the size of a forward ant increases for every hop as address information for the last hop is appended. The size of one unit of address information is 16 bytes. Thus, the size of a forward ant increases linearly with 16 bytes per hop, whereas the size of a flood request is constant. Figure 8.5 shows the relationship between ant and flooding solution message sizes during resource search (backward ant/resource reply not included). The size

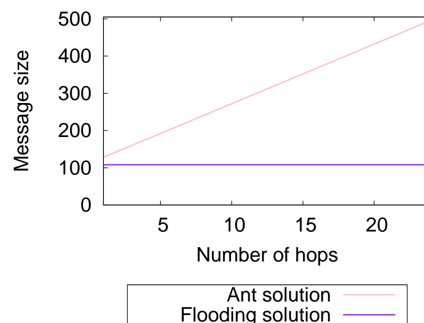


Figure 8.5: Message size during resource search in both solutions plotted against the number of hops traveled.

of a backward ant is the same as a forward ant, as these use the same data structure. The only difference is that the size of the backward ant decreases with the number of hops. A flood resource reply, on the other hand, is only 92 bytes.

Note also that these sizes are including some information needed for testing purposes, and may be removed during normal system use.

8.3.2 Location Learning

Tables 8.6 and 8.7, columns ”Messages Sent” and ”Bytes Sent” respectively, show the number of messages and bytes sent in each of the location learning scenarios. Again, we see that the ant solution benefits from the introduction of location learning, as in all three scenarios, the ant solution performs significantly better than the flooding solution with respect to the number of *messages* sent: The flooding solution nodes send approximately 61% more messages than the ant solution nodes in the dense scenario and 88% more messages in the sparse scenario. However, like in the single localization scenarios we see that the large size of the ant still makes the ant solution send a significantly higher number of *bytes* than the flooding solution.

When comparing the results from the two mobility scenarios, we see that the sparse scenario gives better performance than the dense scenario: Nodes in the ant solution send approximately 18% more messages in the dense than in the sparse scenario. This is probably due to the difference in number of neighbors: Fewer neighbors increases the possibility of choosing the ”correct” neighbor as next hop. For the flooding solution, the numbers are more or less equal.

Scenario	Solution	Messages Sent	Messages Received	Utilization
Grid	Ant	57.0	195.2	29.2
	Flood	88.1	258.8	34.04
M-Dense	Ant	91.9	763.1	12.04
	Flood	148.4	1316.0	11.28
M-Sparse	Ant	77.7	553.0	14.05
	Flood	146.0	813.6	17.94

Table 8.6: Messages sent and received and message utilization for both solutions in each location learning scenario. M-dense and M-sparse are dense and sparse mobility scenarios, respectively.

Scenario	Solution	Bytes Sent	Bytes Received	Utilization
Grid	Ant	14291.2	48908.8	29.22
	Flood	9257.2	27457.6	33.71
M-Dense	Ant	24612.8	193644.8	12.71
	Flood	15572.4	140800.0	11.06
M-Sparse	Ant	19454.4	137593.6	14.14
	Flood	15352.0	86867.2	17.67

Table 8.7: Bytes sent and received and message utilization for both solutions in each location learning scenario. M-dense and M-sparse are dense and sparse mobility scenarios, respectively.

8.3.3 Conclusion

What the bandwidth tests show us is that the ant solution has the potential to outperform the flooding solution when the network size and/or the number of requesting nodes increase. Increasing network size causes a higher degree of flooding, whereas the number of nodes requesting nodes has an impact on how much we get out of the resource location learning. However, in all test scenarios, the large ant size causes the ant solution to behave poorly with respect to the number of bytes sent. As stated in Chapter 6, the ant data structure is not optimized with respect to size, and it should thus be possible to decrease the size of the ants. However, there will always be a need of appending address information to forward ants for the corresponding backward ant to be able to retrace its path. Thus, the size of a forward ant will always have to increase with the number of hops traveled. However, one might consider other, better scalable ways to store this information than the 16 bytes large sockaddr_in structure that is currently used.

Also, in our experiments, the ant time to live was equal to the number of nodes in the network minus one. This means that our ants are allowed to grow from 128 bytes to 496 bytes, or almost four times its own size, in the grid and mobility scenarios consisting of 25 nodes. From our results, we see that the time to live-value should probably be set to a smaller number. By doing this, we limit the maximum size of a forward ant. However, we also make it harder for a forward ant to find the resource during its search, as it is allowed to visit a smaller number of nodes.

Another interesting observation is that if we compare the results from the ant solution response time measurements in number of hops with the corresponding number of sent messages, we see that the latter numbers are only slightly smaller than the former. Remember that the response time is measured from a forward ant is sent from the resource-requesting node to the resource-holding node is reached, whereas the number of sent messages also includes the backward ant. If we did not eliminate loops traveled by the forward ant before sending the corresponding backward ant, we would expect the number of sent messages to be approximately twice as large as the number of hops traveled by the forward ant. However, as any loops are eliminated, the number of hops traveled by the backward ant, and thus the number of sent messages, may be lower than that of the corresponding forward ant.

If we look at the chain scenario with a single requesting node, we see that the number of hops is 4.0 and the number of sent messages is 8.0, exactly twice as much. This fits well with what we expect for this scenario, as the characteristics of the chain scenario makes it impossible for the forward ant to travel in loops. For the grid scenario, however, the number of hops is 21.0, whereas the number of sent messages is 26.7, only approximately 21% larger. Thus, the length of the path traveled by the backward ant back to

the requesting node is drastically reduced. The reason for this is that the forward ant traveled in a long loop, which was later eliminated. The same holds for the mobility scenario, where the number of sent messages is only approximately 9% larger than the number of hops.

This will have an impact on the results we get from location learning. The gain from this concept is highly dependent of the learned location information spreading to as many nodes as possible, rising the possibility for subsequent ants searching for similar resources reaching nodes with resource location information. However, by decreasing the length of the path traveled by the backward ant, we limit the number of nodes picking up the new location information. Thus, from these results, we see that one might want to consider allowing also backward ants to travel in loops.

Loops were eliminated because we wanted to use as little bandwidth and processing power as possible to get the backward ant to the requesting node. Thus, we are dealing with a tradeoff: We can either spend less resources, or we can get a good location information spreading. Which approach to use depends on the application: If it is likely that several subsequent ants look for similar resources, a good location information dissemination could weigh up for the expensive first localization. If, in most cases, only one node will be requesting a certain resource, however, most requests will be expensive first-time requests, and we do not get to utilize the well-disseminated location information.

8.4 Results — Processing Power Usage

Processing power usage is measured in the number of *received* messages, thus the number of messages that, in some way, need to be handled by the receiving node. This includes both messages that are delivered all the way up to the application layer as well as those that are received but only forwarded or discarded at the transport layer.

In the ant solution, we expect the number of sent messages from the application level to be equal to the number of received messages, as all nodes receiving an ant should either re-forward it or, if the ant is a forward ant and the resource is present: Reply with a backward ant. An intuitive thought is that if a forward ant "dies" because of a too low time to live-value, there will only be a received message, not a sent message. However, this will lead to the requesting node sending a new forward ant. This node will only get one reply back, though, thus the extra sent message from the requesting node "eliminates" the extra received message at the node receiving the forward ant with time to live = 0.

However, as mentioned above, these results contain both application layer and network layer messages. As stated earlier, the ant solution does not exploit the fact that all nodes within range of a transmitting node will overhear

the transmitted package and have to check if it is destined for itself or some other node. In the flooding solution, however, all nodes hearing a message pick it up, but only retransmit if an equal message has not already been received and retransmitted.

8.4.1 Single Localization

The results from our single localization experiments are shown in Table 8.4, column "Messages Received" and 8.5, column "Bytes Received".

Chain Scenario

As expected, the number of received messages is equal for both solutions in the chain scenario, as all messages follow the exact same path. As observed in the above bandwidth experiments, the number of bytes is quite a bit larger for the ant solution than for the flooding solution, again because of the much larger ant solution message size.

Grid Scenario

In the grid scenario, the two solutions perform approximately equally; the ant solution receives on average 0.3 messages less than the flooding solution. As the network size and number of hops needed increases, the number of bytes also increases, and the difference between the ant solution and the flooding solution is even larger than in the chain scenario.

Mobility Scenario

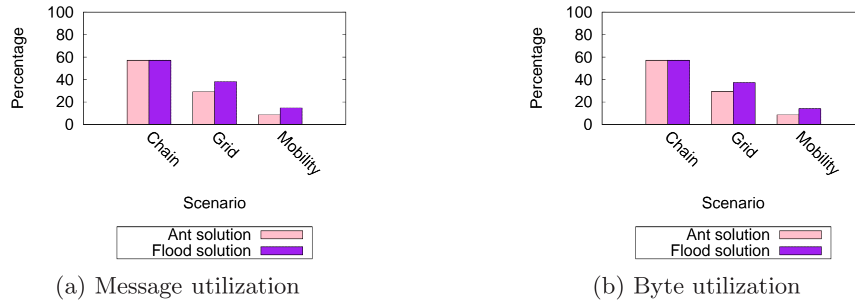
In the mobility scenario, however, the number of received messages in the ant solution increases dramatically, and is approximately 2.4 times higher than that of the flooding solution. This is also shown in the number of bytes received, and in this scenario, the nodes in the ant solution receives approximately 6.5 times as many bytes as the nodes in the flooding solution. The large increase is due to the large number of neighbors receiving everything transmitted from their neighbors.

8.4.2 Location Learning

The results from the location learning experiments are listed in Table 8.6 and 8.7 under the columns "Messages Received" and "Bytes Received" respectively.

Grid Scenario

The flooding solution here uses approximately 32.6% more messages to localize the requested resources. When measuring in number of bytes, the



(a) Message utilization

(b) Byte utilization

Figure 8.6: Total utilization percentage in each scenario.

ant solution is still outperformed by the flooding solution, as the ant solution nodes receive approximately 78% more bytes than the flooding solution nodes.

Mobility Scenario

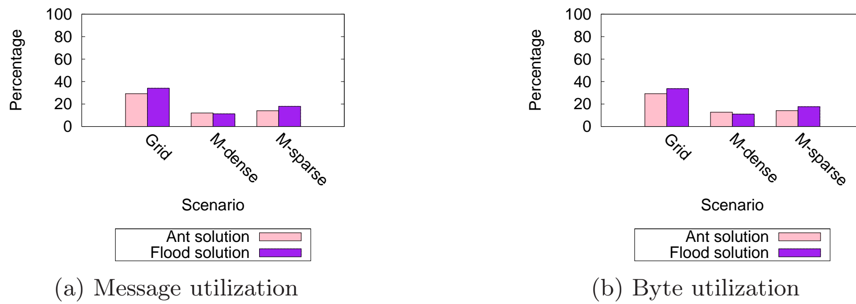
In the dense scenario, the flooding solution nodes receive 72.5% more messages than the ant solution, but still the ant solution nodes receive 37.5% more bytes. In the sparse scenario, the difference is smaller, but the flooding solution still receives approximately 47% more messages. We also see the same as in the bandwidth usage experiments: The ant solution performs better in the sparse scenario than in the dense: Nodes in the dense scenario receive approximately 38% more messages than nodes in the sparse scenario.

8.4.3 Conclusion

In these experiments, we see the same tendency as in the bandwidth experiments: When introducing more requesting nodes, the ant solution localizes the requested resource using less resources than the flooding solution, given that we measure in number of messages. Still, the ant size makes the ant solution perform quite a bit worse than the flooding solution with respect to the number of bytes received. We also see that the ant solution benefits from having a low number of neighbors, as for local processing powers, a lower number of neighbors means a lower number of overheard messages.

8.5 Results — Processing Power Utilization

The local processing power utilization is the ratio of sent messages to received messages, and expresses the percentage of received messages that a node found worth either forwarding or replying to, meaning the messages that the node received and actually used to contribute to the resource localization.



(a) Message utilization

(b) Byte utilization

Figure 8.7: Total utilization percentage in each location learning scenario. M-dense and M-sparse are dense and sparse mobility scenarios, respectively.

8.5.1 Single Localization

Figure 8.6 shows utilization of messages (8.6a) and bytes (8.6b). The actual percentages are also listed in Tables 8.4 and 8.5.

In the single localization scenarios, we see that the flooding solution generally performs equal to (chain scenario) or better (grid and mobility scenarios) than the ant solution. The difference in the message utilization percentages is not very large, however: 8.92% and 6.19% in the grid and mobility scenarios respectively. The byte utilization results are very similar to the corresponding message utilization percentages, which is natural because we are dealing with the same actual messages in both cases.

This corresponds well with what we expected, as nodes in the ant solution receive all messages sent from neighboring nodes, but only utilize those addressed to themselves. In the flooding solution, however, all messages are handled if previously not received and utilized.

8.5.2 Location Learning

Figure 8.7 shows utilization of messages (8.7a) and bytes (8.7b). Percentages are also given in Tables 8.6 and 8.7.

For the grid scenario, we see the same tendency as in the single localization scenarios: The flooding solution achieves a higher utilization percentage than the ant solution. This is also the case for the sparse mobility scenario. In the dense mobility scenario, however, the ant solution performs slightly better than the flooding solution and achieves a 0.76 percent higher message utilization.

We also see that the utilization of local processing resources is slightly higher in the sparse than the dense mobility scenario, especially for the flooding solution. Again, this is due to the lower number of neighbors in the spare scenario — fewer nodes receive messages that are not meant for them (ant solution) or that have already been received from other nodes (flooding solution).

8.5.3 Conclusion

The differences in local processing power utilization are not very large, and seem to decrease with the size of the network and number of neighbors. This is natural, as the number of flooded messages will increase drastically when the total number of nodes and the number of neighbors per node increases. Under such circumstances, it will be harder for the forward ant to actually locate the requested resource in a larger network, but the mere fact that the network grows will not have an impact on the utilization, but only the number of hops needed to localize a resource, and thus also the number of sent and received messages. However, if the average number of neighbors per node increases, more messages will be unnecessarily overheard, lowering the utilization.

Chapter 9

Conclusion and Further Work

> *"A conclusion is simply the place where someone got tired of thinking."*
>
> — *Arthur Block*

In this last chapter we provide a summary of this thesis and its contribution. We also give a summary of the performance analysis and observations done during performance testing and analysis and discuss some possible solutions and improvements to some of the observed problems and flaws. We also take a step back and look at the entire process from beginning to end. At last, we look at how our work may be continued, both in terms of further testing, to learn more about the solution's performance, as well as how the solution may be extended.

9.1 Contribution

The purpose for this thesis has been to design and develop a system enabling nodes in need of a resource to autonomously localize this resource if present within the same partition of a MANET. The system is proposed as a solution if no other framework for resource location information sharing and dissemination is present, and is thus a general-purpose solution, not tailored to function with any specific applications. When a node in the MANET finds that it needs a certain resource not present within the node itself, it uses our solution to issue a resource request. If the resource is present at some node within the same network partition, the requesting node should, eventually, get a resource reply from this node.

The characteristics of MANETs make design and implementation harder, as one always needs to consider the low availability of resources, such as bandwidth and processing power, as well as the (possibly) ever-changing

network topology. Our aim has thus been to develop a system that required as little of these resources as possible rather than lowering the system's response time.

Our system should be able to function autonomously, and should thus preserve the *self-* properties*, such as self-organizing, self-configuration, self-optimization, self-protection and self-healing. Systems with such characteristics have always existed in nature, and may be used as inspiration when designing computer systems. In this thesis we have looked at *stigmergy* — the foraging behavior of *ants*, who communicate indirectly with one another by leaving pheromone trails on the ground. These trails, together with the fact that ants tend to prefer paths with high pheromone concentration, are used to make most ants walk the same, most optimal path between their nest and a food source when gathering food.

The difference between our system and ants, is that we want our artificial ants to spread out through the network to be able to search every corner for the requested resource. To do this, we have turned ants' behavior around and designed a system using what we call *opposite stigmergy*. In this approach, instead of walking the most used, or most recently used, path from a resource-requesting node, we choose the least recently used path.

In addition to the ant solution, we have designed and developed a very simple flooding-based solution. The goal for this solution is the same as for the ant solution: To localize a given resource — if present — within the MANET partition. However, the characteristics of flooding makes this approach behave and perform quite different from the ant approach. What we expected was low response times but high resource usage. The aim of this solution was to provide alternative performance results for comparison with the ant solution.

We have tested our two solutions in a range of test scenarios and analyzed their performance in terms of response time, bandwidth usage and local processing power usage and utilization.

9.2 Performance Evaluation

The ant solution is designed according to two requirements. The first is an absolute requirement:

- If a requested resource is present within the network, it must eventually be found.

The other requirement reflects the desired performance and relates to the environment in which the solution is designed to run, namely MANETs:

- The solution should consume as little resources, network wide as well as node local, on one resource localization.

- The solution should have a good resource utilization, both with respect to transmissions and local processing power.

Although low response time is also an advantage, this is, because of MANET characteristics, considered less important than good resource utilization.

Metrics were developed according to these requirements, and all results from the ant solution have been compared to the corresponding results from the flooding solution to see if the ant solution is actually worth using over other, simpler solutions.

From our test results we see that in many cases, the flooding solution both has a lower response time as well as a lower resource usage. However, we also see that when introducing more than one resource-requesting node, both the response time and the number of sent and received messages for the ant solution decreases drastically. As expected, the ant solution did not perform better than the flooding solution in any of the response time test cases, but some of this is due to the much larger amount of processing needed on each node in the ant solution. Data structures and algorithms can probably be optimized to lower this delay somewhat, but there will always be a need for more extensive processing in the ant solution than in the flooding solution.

If we look at the number of messages sent (bandwidth usage) and received (local processing power usage) in the two location learning test scenarios, we see that the ant solution outperforms the flooding solution, thus, in number of messages, we have managed to get a lower resource usage than that of a flooding system, which was our primary goal. If we measure in number of bytes, however, the large size of the virtual ants makes the ant solution perform worse than the flooding solution. Again, it it possible to optimize the data structures used, and thus lowering the amounts of bytes sent and received. As stated in Section 8.2.3, this also tells us that our time to live-limit was too high — using a lower limit might have helped, but might of course also cause the need of more tries to be able to locate a resource. Thus, more tests are needed to figure out how small we may set this value without introducing this new problem instead.

From the mobility scenario test cases, we also learned that a too high number of neighboring nodes makes it too hard for the ant to choose the "correct" way — it needs too many tries to choose the set of neighbors that actually leads to the resource.

9.3 Critical Assessments

When we first started the work with this thesis, a lot of time was spent searching for information regarding similar work. A lot of information is available on how stigmergy may be used for routing, both in traditional,

wired networks as well as in MANETs, and also for a handful of other problems related to optimization.

In the beginning, we imagined that our approach would be fairly similar to the existing ACO approaches. The deeper into the problem we dove, however, the more we realized that our problem was, in some senses, fundamentally different from that of the algorithms and approaches we had studied to gather background information. We realized that what we were looking for was the *opposite* of anything we had seen — we wanted our artificial ants to *spread* throughout the network, not to *gather* themselves on one path. In some areas, this lead to a lot of thinking, experimenting, failing and more thinking, whereas in other areas, it simplified things a lot. For example, our pheromones are, at least at this stage, reduced to a timestamp, providing built-in pheromone evaporation. The hardest part was thus in many cases to realize the solutions and to see, and also to *accept*, their simplicity, not the actual implementation of the system.

The test setup and actual test phases were cumbersome. The lack of experience with emulators such as NEMAN, together with a somewhat unruly test computer, slowed the process down a bit, but we did manage to run the most important test cases, although we would have liked to get a more complete set of test results.

We see from our performance analysis that there are a handful of things that probably could have been done better another way. We also see that our results do not paint the whole picture — more tests are needed to clarify exactly which scenarios may benefit from our solution and which will probably not. However, we do see a clear tendency and a potential that in certain scenarios and under certain circumstances, an ant approach to the resource localization problem in MANETs is feasible.

Had we known when we started what we know now, a lot of problems and design choices would obviously have been a lot easier and less time consuming. However, this is part of the process: Designing, implementing, testing, analyzing and then starting all over again, trying to address the problems discovered when analyzing.

9.4 Further Work

9.4.1 Resource Localization in Sparse Networks

The solution developed in this thesis only works for dense networks, i.e. consisting of one partition only. However, MANETs are often sparse. In some situations, the network may consist of more than one partition. Our solution will then only search for a requested resource within the partition containing the requesting node.

If a resource satisfying the demand is located within the partition, there is probably no reason to start searching in other partitions. It is probably

much easier for the requesting node to utilize the resource if it is located in the same partition. However, the requesting node may be able to adapt if the only available resource is located in another partition, for example by moving to that partition itself, or by utilizing delay tolerant techniques as described in Section 2.1.3.

A natural expansion of our system is thus to enable searching in other partitions. To do this, we need a way to get the request and any replies to other partitions and back. We imagine an approach inspired by for example epidemic routing, where nodes could carry forward ants for a while, and if no corresponding backward ant is received, the request could be forwarded if the node should meet new nodes. The difficult part here is figuring out which new nodes to forward the request to and not, as it is not necessary to forward the request to nodes in the same partition as the requesting node. Also, we need a way to stop the whole process if a resource is found, which is difficult when nodes in several partitions may be working on solving the same problem, but are unable to (easily) communicate with each other.

9.4.2 Resource Goodness

The existing solution does not differ between resource qualities as long as they fulfill the demands of the requesting node. The "goodness" of a resource can reflect the quality measurement of the resource, but also for example how close to the requesting node the resource holder is currently placed, how stable the resource is, both in terms of resource concentration (if appropriate for the given resource), node mobility and node up-time, or perhaps the load, both on the path between requesting and holding nodes and on the actual resource on the holding node. If several nodes hold the requested resource, it might thus be good to locate several resource-holding nodes and pick the one that best satisfies the needs of the requesting node.

Locating several resources will put a higher load on the network, and should thus only be done in MANETs that can deal with this extra cost. However, choosing for example the resource-holding node closest to the requesting node might save resources if the requesting node decides to actually utilize the located resource in some way.

If we introduce some sort of goodness, this needs to be reflected in the pheromone values, so that subsequent ants can use this information when looking for a given resource. As mentioned in Section 5.5.5, there is a fundamental difference in the information given by our pheromone values and the information provided by a goodness value: A goodness value is information on a resource that has *actually been localized*, whereas our pheromone values provide information about where ants have *looked for* a resource. One thus needs to find a way to combine these two values.

One should also consider keeping track of *every* located resource, not only the best like the current solution does. It is then possible to for example

return the lowest quality matching resource if several are available, saving better resources for those that actually demand such high quality.

9.4.3 Further Implementation, Testing and Analysis

The tests and analyses from Chapter 8 give an indication of how the ant solution works, but are not enough to finally conclude on the performance of the system. More tests are needed, and in addition to this, from the performance analysis we have learned a few things about which changes should be made to the system in order to increase its performance.

- In our grid and mobility scenarios, there are 25 nodes, one of which is the node holding the requested resource. The very low resource density makes the probability that an ant stumbles upon this node quite low — the ant needs quite a bit of luck to locate the resource. This is reflected by our test results, where we see that a forward ant needs a very high number of hops to locate a resource in the two larger scenarios. More resource-holding nodes increases the possibility that a forward ant stumbles upon a satisfying resource. Tests should thus be performed with more than one resource-holding node to see how this impacts the performance of the system.

- There are several parameters that may be tuned to the network characteristics, such as forward ant time to live-values and sleep time between two subsequent forward ants. Extensive testing may be needed to find out the best ways to set these values. For example, it might be a good idea to start with a quite low time to live-value, and if no resource is located, increase this value incrementally until a resource is found.

- As stated in Section 8.3.3, one should consider allowing loops to get a better resource location information dissemination. An alternative is allowing for example loops up to a certain length to avoid wasting too much resources on very long loops.

- Tests should be performed to see how many neighbors nodes may have on average before this lowers the system performance.

- We should consider keeping pheromone lists for a neighbor for a while after it is lost in case it comes back.

- Implement an alternative ant solution which combines a reactive and proactive approach to scheduling as explained in Section 5.5.1.

- We should consider making the ant solution more robust to transmission errors, as explained in Section 8.1.1. However, if such error handling introduces too much overhead to the system, it might be best

to skip this. As long as the resource-requesting node does not get a backward ant back, it will continue searching, so the resource will still eventually be found. Whether this should be handled or not also depends on the level of mobility in the MANET — with relatively low node mobility, the chance of sending a forward ant to a very recently lost neighbor is quite small.

Bibliography

[1] olsrd - an adhoc wireless mesh routing daemon. Available [online] `http://www.olsr.org/` (Retrieved 22.03.2010).

[2] TCPDUMP/LIBCAP public repository. Available [online] `http://www.tcpdump.org/` (Retrieved 22.03.2010).

[3] Christian Blum and Xiaodong Li. Swarm intelligence in optimization. In Christian Blum and Daniel Merkle, editors, *Swarm Intelligence*, Natural Computing Series, pages 43–85. Springer, 2008.

[4] G. Di Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *J. Artif. Intell. Res. (JAIR)*, 9:317–365, 1998.

[5] Gianni Di Caro, Frederick Ducatelle, and Luca Maria Gambardella. Swarm intelligence for routing in mobile ad hoc networks. In *In Proceedings of the 2005 IEEE Swarm Intelligence Symposium (SIS*, 2005.

[6] Daily News From Cornell University Cornell Chronicle Online. Chemical helps ants remember where they left their food, shows promise for alzheimer's disease, cornell scientists report, 1998. Available [online] `http://www.news.cornell.edu/releases/Feb98/antpheromone.hrs.html` (Retrieved 31.03.2010).

[7] A. K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[8] Gianni Di Caro, Frederick Ducatelle, and Luca Maria Gambardella. Anthocnet: an ant-based hybrid routing algorithm for mobile ad hoc networks. In *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 461–470, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[9] M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant algorithms and stigmergy. *Future Gener. Comput. Syst.*, 16(9):851–871, 2000.

[10] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[11] Thomas D. Dyer and Rajendra V. Boppana. A comparison of TCP performance over three routing protocols for mobile ad hoc networks. In *MobiHoc*, pages 56–66. ACM, 2001.

[12] Erling Falck-Ytter. Brannen i majorstutunnelen. Available [online] http://arkiv.brannmannen.no/?cid=11&aid=44980/ (Retrieved 04.03.2010).

[13] Laura Marie Feeney. A taxonomy for routing protocols in mobile ad hoc networks. Technical Report T99-07, Swedish Institute of Computer Science, November 1, 1999.

[14] Dina Q. Goldin and David Keil. Toward domain-independent formalization of indirect interaction. In *WETICE*, pages 393–394. IEEE Computer Society, 2004.

[15] Marc Greis. Marc greis' tutorial for the UCB/LBNL/VINT network simulator "ns", Chapter XI: Generating traffic-connection and node-movement files for large wireless scenarios. Available [online] http://www.isi.edu/nsnam/ns/tutorial/nsscript7.html (Retrieved 24.03.2010).

[16] International Organization for Standardization. *Programming Language – C*. ISO/IEC 9899.

[17] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York, NY, 1991.

[18] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[19] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS*, pages 96–107, 2002.

[20] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[21] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[22] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. Mobinet: a scalable emulation infrastructure for ad hoc and wireless networks. In *WiTMeMo '05: Papers presented at the 2005 workshop on Wireless traffic measurements and modeling*, pages 7–12, Berkeley, CA, USA, 2005. USENIX Association.

[23] S. Mehfuz and M. N. Doja. Swarm intelligent power-aware detection of unauthorized and compromised nodes in manets. *J. Artif. Evol. App.*, 2008:1–16, 2008.

[24] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162, New York, NY, USA, 1999. ACM.

[25] Hean Kuan Ong, Hean Loong Ong, Elok Robert Tee, and R. Sureswaran. Scalability study of ad-hoc wireless mobile network routing protocol in sparse and dense networks. In *Distributed Frameworks for Multimedia Applications, 2006. The 2nd International Conference on*, pages 1 –8, may 2006.

[26] Grassé P. La reconstruction du nid et les coordinations inter-individuelles chez *Bellicostitermes natalensis* et *Cubitermes sp.* La théorie de la stigmergie: Essai d'interpretation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[27] Margie Patlak, Thomas Baker, May Berenbaum, Ring Cardé, Thomas Eisner, Jerrold Meinwald, Wendell Roelofs, and David Wood. Insect pheromones: Chemicals of communication. *Beyond Discovery*, 2003.

[28] J. Postel. RFC 768: User datagram protocol, August 1980.

[29] J. Postel. RFC 793: Transmission control protocol, September 1981.

[30] Matija Pužar. *Towards Secure and Reliable Information Sharing in Emergency and Rescue Operations.* PhD thesis, University of Oslo, 2010.

[31] Matija Pužar and Thomas Plagemann. NEMAN: A network emulator for mobile ad-hoc networks. Technical report, Department of Informatics, University of Oslo, 2005.

[32] David Rising. Battlefield internet helps forces in iraq, April 2003. Available [online] `http://www.redorbit.com/news/technology/705/battlefield_internet_helps_forces_in_iraq/index.html` (Retrieved 04.03.2010).

[33] Elva J.H. Robinson, Duncan Jackson, Mike Holcombe, and Francis L.W. Ratnieks. Insect communication: 'no entry' signal in ant foraging. *Nature*, 438(7067):443, 11 2005.

[34] S. Schmid, M. Sifalakis, and D. Hutchison. Towards autonomic networks. In *Autonomic Networking, First International IFIP TC6 Conference, AN 2006, Paris, France, September 27-29, 2006, Proceedings*, volume 4195 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2006.

[35] G. Di Marzo Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi. Dependable self-organising software architectures - an approach for self-managing systems. Technical report, School of Computer Science and Information Systems, Birkbeck College, London, 2006.

[36] G. Di Marzo Serugendo, M. P. Gleizes, and A. Karageorgos. Self-organisation and emergence in MAS: An overview. *Informatica (Slovenia)*, 30(1):45–54, 2006.

[37] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, **2**:333, 1994.

[38] Jun-Zhao Sun. Mobile ad hoc networking: An essential technology for pervasive computing. In *Proc. International Conferences on Info-tech & Info-net, Beijing, China*.

[39] A. Toninelli. *Semantic-Based Middleware Solutions to Support Context-Aware Service Provisioning in Pervasive Environments*. PhD thesis, Università di Bologna, 2008.

[40] USC/ISI UC Berkeley, LBL and Xerox PARC. The network simulator - ns-2. Available [online] `http://www.isi.edu/nsnam/ns/` (Retrieved 06.04.2010).

[41] Amin Vahdat and David Becker. Epidemic routing for partially-connected ad hoc networks, May 08 2000.

[42] Tristram D. Wyatt. *Pheromones and Animal Behaviour: Communication by Smell and Taste*. Cambridge University Press, 2003.

[43] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. *SIGSIM Simul. Dig.*, 28(1):154–161, 1998.

[44] Wenrui Zhao, Mostafa H. Ammar, and Ellen W. Zegura. A message ferrying approach for data delivery in sparse mobile ad hoc networks. In Jun Murai, Charles E. Perkins, and Leandros Tassiulas, editors, *MobiHoc*, pages 187–198. ACM, 2004.

[45] Jim Zyren and Al Petrick. IEEE 802.11 Tutorial, 2002. Available [online] `http://www.techonline.com/learning/techpaper/193101922` (Retrieved 04.03.2010).

Appendices

Appendix A

A Sample Scenario File

This chapter provides a quite simple sample scenario file. The different components of the scenario file is explained in comments within the file. Please note that, for aesthetic purposes, some lines have been split. This, however, would not be accepted by NEMAN.

A.1 Chain Scenario

The following is a simple chain scenario file where the node with IP address 10.0.0.1, port 3457, is instructed to search for a resource with name *memory* and quality 90 at 10 seconds into the emulation.

```
#
# nodes: 5, pause: 0.00, max speed: 0.00,
  max x: 300.00, max y: 300.00
#

# Set node locations:
$node_(0) set X_ 10.000000000000
$node_(0) set Y_ 10.000000000000
$node_(0) set Z_ 0.000000000000
$node_(1) set X_ 60.000000000000
$node_(1) set Y_ 10.000000000000
$node_(1) set Z_ 0.000000000000
$node_(2) set X_ 110.000000000000
$node_(2) set Y_ 10.000000000000
$node_(2) set Z_ 0.000000000000
$node_(3) set X_ 160.000000000000
$node_(3) set Y_ 10.000000000000
$node_(3) set Z_ 0.000000000000
$node_(4) set X_ 210.000000000000
$node_(4) set Y_ 10.000000000000
```

```
$node_(4) set Z_ 0.000000000000

# Set which nodes are within range of each other:

$god_ set-dist 0 1 1
$god_ set-dist 1 2 1
$god_ set-dist 2 3 1
$god_ set-dist 3 4 1

# Needed to make the test last for 20 seconds
$ns_ at 20.0 "$god_ set-dist 0 1 1"

# Send a request for resource memory of quality 3 to tap1:
$ns_ at 10.000000000000 "$node_(10.0.0.1)
 sendmsg port=3457 msg='memory 3'"


# Set broadcast range to get a
# correct view in the NEMAN gui:
# broadcast_range: 75.00
# data_range: 75.00
```

Appendix B

The Flooding Solution

The basics of the flooding solution were explained in Section 7.4. This chapter provides a more detailed explanation of parts of the source code and functionality of the flooding solution.

B.1 Data structures

Most of the flooding solution program is built on the same code as the ant system. Resources are the same, although the parts not needed by the flooding solution have been removed. However, a couple of new data structures have been added:

The **flood_package** structure, shown in Figure B.1, is the structure holding the resource request. The **resource** structure is the same as the one used in the ant system, described in Section 6.7.2. The **seq_nr** field is used to make sure no node broadcasts the same resource request more than once. hop_count and time_of_request are used for performance testing purposes only.

Each node keeps a list **sequence_numbers** consisting of {node, sequence number} couples, more specifically **node_seq** structures, as shown

```
struct flood_package {
  char type;
  struct resource res;
  struct sockaddr_in requesting_node;
  short seq_nr;
  short hop_count;
  struct timeval time_of_request;
};
```

Figure B.1: Flood_package structure.

```
struct node_seq {
  struct sockaddr_in *node;
  short highest_seen_seq;
};
```

Figure B.2: Node_seq structure.

```
struct request_answer {
  char type;
  struct resource res;
  struct sockaddr_in requesting_node;
};
```

Figure B.3: Request_answer structure.

in Figure B.2. The list contains one entry for every other node this node has ever heard of and the highest seen sequence number for this node. If a request with sequence number less than or equal to this number for the corresponding node, it is thrown away. If requests should, for some reason, appear out of order, this functionality may make a node throw away requests that should have been re-broadcast. In this very simple solution, however, we have chosen not to handle this, as this should not happen very often.

The last new structure is the *request_answer* structure, shown in Figure B.3. This structure is used for answers to requests.

B.1.1 Logging

Logging is done as in the ant solution, see Section 6.5 for details.

B.1.2 Program Flow

Threads

As the program should be able to manage several tasks at once, it is divided into several threads:

- *The main process:* Listens for incoming resource requests and resource replies.
- *Thread 1:* Listens for incoming resource requests. Created by the main process shortly after startup.
- *Thread 2:* Initiates a resource search. Created by Thread 1 for every incoming resource request.

Appendix C

Source Code

This chapter will go through some of the source code for the ant solution.

C.1 Program Layout — Ant Solution

The source code is divided into a number of files with respect to each functions purpose. These files are:

- **main.c** Initialization, reception of incoming resource requests and search initiation.
- **ant.[ch]** The ant structure, all functions regarding ant sending, receiving and handling.
- **neighbor.[ch]** The neighbor and neighbor_prob structures, topology update reception and neighbor adding/deleting.
- **resource.[ch]** The resource and resource_info structures, resource information registry and lookup.
- **pheromone.[ch]** The pheromone structure, building and updating of pheromone lists.
- **communication.[ch]** All functions used to handle sockets, establishing connections and own address information retrieval.
- **util.[ch]** Type definitions and various utility functions, such as list operations, log file operations and uniformly distributed random number generation.

C.2 Program Layout — Flooding Solution

This program is quite a bit smaller than the ant solution, but is still divided into several files:

- **flood_main.c** Initialization, reception of incoming resource requests and search initiation.
- **flood.[ch]** The resource request and resource reply structures, all functions regarding request sending, receiving and handling.
- **resource.[ch]** The resource and resource_info structures, resource information registry and lookup.
- **communication.[ch]** All functions used to handle sockets, establishing connections and own address information retrieval.
- **util.[ch]** Type definitions and various utility functions, such as list operations and log file operations

Most of the code, unless the code in flood_main.c and flood.[ch] is more or less identical to the corresponding ant solution code. Redundant functions and variables have, however, been removed.

C.3 Building the Source Code

In addition to the source code, there are also makefiles (one for each solution) which may be used to compile the source code. There are two options when building the solution: Running make with the default target and with the "test" target. Building with the test target makes the system produce an additional log file when run, namely the *ant_test.log*/*flood_test.log* file. This file is written to whenever a request for a resource either present at the receiving node or with a known location is received. The information logged includes receiving node, requesting node, time spent on localization (in milliseconds, requiring synchronized clocks) and the number of hops traveled by the forward ant/resource request, loops included.

C.4 Running the Source Code

The ant application may be run with the following command:

```
./ant <device> <ant_ttl> <max_sleep_time>
```

where *device* is the name of the network interface the program should use, *ant_ttl* is the number of hops one ant should be allowed to travel, and *max_sleep_time* is the time one search may last before a new search is initiated.

The flooding application is run with the following command:

```
./flood <device>
```

where *device* is the name of the network interface the program instance should use.

Appendix D

Code Examples

This chapter shows the source code of some of the most important parts of the ant solution, namely the sending and receiving of ants. Please note that all code concerned with test information and debug information has been left out from this listing.

D.1 send_forward_ant()

send_forward_ant() is the function that is used to send forward ants from neighbor to neighbor. It takes a ready made ant structure as argument, registers the requested resource in the list of all resources ever heard of, chooses a next hop neighbor and sends the forward ant there.

```
/* This function takes a forward ant as argument, picks a neighbor and
 * forwards the forward ant to this neighbor */
void send_forward_ant(struct ant *f_ant)
{
  int sockfd = get_unbound_udp_socket(iface);

  register_resource((f_ant)->res);

  struct sockaddr_in *ant_source = NULL;
  if(f_ant->preceding_nodes > 0)
  {
    ant_source = (struct sockaddr_in *) (((char *)f_ant)
               + sizeof(struct ant)
               + (f_ant->preceding_nodes - 1)
               * sizeof(struct sockaddr_in));
  }
  /* Find a neighbor to forward the ant to */
  struct neighbor *n = get_neighbor(&(((struct ant *)f_ant)->res),
                                    ant_source);
  if(n == NULL)
  {
    free(f_ant);
    close(sockfd);
    return;
  }
  struct sockaddr_in *si_dest = n->si_n;
```

```
  memcpy(&f_ant->next_hop, si_dest, sizeof(struct sockaddr_in));
  si_dest->sin_port = htons(atoi(ANT_PORT));

  /* Calculate size of new ant and forward */
  int ant_size = sizeof(struct ant)
               + ((struct ant *)f_ant)->preceding_nodes
               * sizeof(struct sockaddr_in);

  int sent = sendto(sockfd,
                    f_ant,
                    ant_size,
                    0,
                    (struct sockaddr *)si_dest,
                    (sizeof(struct sockaddr_in));

  if(sent == -1)
  {
    close(sockfd);
    free(f_ant);
    return;
  }

  close(sockfd);
  free(f_ant);
}
```

D.2 receive_ants()

receive_ants() listens for incoming ants and calls new functions for further handling:

```
/* This function listens for incoming ants,
 * forward as well as backward */
void * receive_ants()
{
  struct sockaddr_in si_src;
  size_t addr_len = sizeof(si_src);
  socklen_t * __restrict al;
  al = (socklen_t *)&addr_len;

  char ant[MAXANTLEN];

  while(1)
  {
    int rv = recvfrom(receive_ants_sockfd,
                      ant,
                      MAXANTLEN,
                      0,
                      (struct sockaddr *)&si_src,
                      al);
    if(rv == -1)
    {
      continue;
    }

    /* Check that ant is actually large enough
     * to contain an ant */
    struct ant *rcvd_ant = (struct ant *)ant;
    if(rv >= sizeof(struct ant) &&
```

```
        rv >= (sizeof(struct ant) +
          rcvd_ant->preceding_nodes * sizeof(struct sockaddr_in)))
    {
      /* Check if ant was destined for this node.
       * Only needed during NEMAN emulation. */
      if(rcvd_ant->next_hop.sin_addr.s_addr !=
          get_own_addr().sin_addr.s_addr)
      {
        continue;
      }

      handle_received_ant(rcvd_ant, si_src);
    }
  }
  close(receive_ants_sockfd);
}
```

D.3 handle_received_ant()

handle_received_ant() is called by **receive_ants()**. It checks the type of the incoming ant and calls functions for further handling based on the ant type. This function is also responsible for appending the address information for the last hop source to the forward ant (if the ant is actually a forward ant).

```
{
/* This function checks if the incoming ant was a forward or backward
 * ant, and creates a new thread to further handle the incoming ant */
void handle_received_ant(struct ant *ant_to_handle,
                         struct sockaddr_in si_src)
{
  pthread_t tid;

  if(ant_to_handle->type == 'F')
  {
    /* Get new thread safe ant with updated address info */
    struct ant * ts_f_ant =
        (struct ant *)add_prev_address_info(ant_to_handle, si_src);
    pthread_create(&tid, NULL, handle_forward_ant, (void *)ts_f_ant);
  }
  else if(ant_to_handle->type == 'B')
  {
    /* Make threadsafe copy of the ant */
    struct ant *ts_b_ant = malloc(sizeof(struct ant)
                           + (ant_to_handle->preceding_nodes
                           * sizeof(struct sockaddr_in)));

    memcpy(ts_b_ant,
           ant_to_handle,
          (sizeof(struct ant) + (ant_to_handle->preceding_nodes
                              * sizeof(struct sockaddr_in))));

    pthread_create(&tid, NULL, handle_backward_ant, (void *)ts_b_ant);
  }
}
```

D.4 handle_forward_ant()

handle_forward_ant() is called by **handle_received_ant()**. It checks if the requested resource is present. If not, it checks if any resource information for the requested resource is present. If not, the ant's time to live value is checked. If more hops are still allowed, **send_forward_ant()** is called for forwarding to a neighbor.

```
/* This functiontakes a forward ant as argument and checks if
 * either the requested resource is present, location information
 * is present or the node should be forwarded to a neighbor */
void * handle_forward_ant(void *ant)
{
  struct ant *f_ant = (struct ant *)ant;
  struct resource *res = lookup_resource(f_ant->res);

  if(res != NULL)
  {
    produce_backward_ant(res, f_ant);
    free(res);
  }
  else
  {
    struct resource *res = lookup_resource_location_info(f_ant->res);
    if(res != NULL)
    {
      produce_backward_ant(res, f_ant);
    }
    else
    {
      if(f_ant->ttl == 1) /* No more hops allowed */
      {
        free(f_ant);
        return NULL;
      }
      /* Forward the forward ant to one neighbor */
      send_forward_ant(f_ant);
    }
  }
  return NULL;
}
```

D.5 handle_backward_ant()

handle_backward_ant() is called by **handle_received_ants()**. It checks whether the backward ant was a reply to itself or if it should be forwarded according to the address information in the ant.

```
void * handle_backward_ant(void *ant)
{
  struct ant * b_ant = (struct ant *)ant;
  register_resource_location_info(b_ant->res);

  if(b_ant->preceding_nodes == 0)
  {
    register_resource_as_located(b_ant->res);
```

```
  }
  else
  {
    /*Forward ant according to ant information */
    forward_backward_ant(b_ant);
  }
  return NULL;
}
```

Appendix E

CD Contents

This chapter describes the contents of the CD appended to this thesis. The CD consists of three folders as described below.

E.1 /implementation

The implementation folder contains the C source code for both solutions.

E.1.1 /implementation/ant_solution

This folder contains the entire C code for the ant solution. The code is organized as explained in Section C.1, and may be compiled as explained in Section C.3 and run as explained in Section C.4.

E.1.2 /implementation/flooding_solution

This folder holds the entire source code for the flooding solution. The organization of the code is explained in Section C.2. The code is built as explained in Section C.3 and run as explained in Section C.4.

E.2 /test_setup

The test_setup folder contains the tools used during the test phase. The root folder contains the following files:

- **change_range.py:** Removes all links with range 2 from a given scenario file, as explained in Section 7.8.2.
- **delay_mobility.py:** Delays all mobility in a given scenario file for 30 seconds, as explained in Section 8.1.1.
- **remove_links.py:** Removes links from a given scenario file with a probability of 0.7, as explained in Section 8.1.3.

- **run_neman.sh:** Runs a given scenario in NEMAN with the -no-gui option.
- **run_nodes_ant.sh:** Runs the ant solution with the given start and end nodes (tap device number), time to live value and max_sleep_time.
- **run_nodes_flood.sh:** Runs the flooding solution with the given start and end nodes (tap device number).
- **run_tests.sh:** Runs a complete set of tests for a given solution. Handles running of the three other run_*-scripts. Takes the following parameters: number of repeated tests, name of scenario file, number of nodes in scenario, name of output folder, and if ant solution also time to live-value and max_sleep_time.
- **tunnel.py:** Needed when running emulations to tunnel control messages from the NEMAN gui to the nodes.

E.2.1 /test_setup/scenarios

This folder contains the scenario files used during testing. The scenarios are:

- **chain.sim:** Chain scenario with one resource-requesting and one resource-holding node.
- **grid.sim:** Grid scenario with one resource-requesting and one resource-holding node.
- **grid_3.sim:** Grid scenario with three resource-requesting nodes and one resource-holding node.
- **mobility.sim:** Mobility scenario with one resource-requesting and one resource-holding node.
- **mobility_5_dense.sim:** Dense mobility scenario with five resource-requesting nodes and one resource-holding node.
- **mobility_5_sparse.sim:** Sparse mobility scenario with five resource-requesting nodes and one resource-holding node.

E.3 /analysis

The analysis folder contains the scripts used for analyzing the data output from the tests. The following files were used:

- **find_avg_hops.py:** Parses a given <solution>.log file and prints the average number of hops and milliseconds used to localize a resource in tests with only one resource-requesting node.

- **multiple_find_avg_hops.py:** Parses a given <solution>.log file and prints the average number of hops and milliseconds per requesting node used to localize a resource in tests with several resource-requesting nodes.

- **find_connectivity.py:** Parses a given scenario file and computes the adjacency matrix and average number of neighbors per node per second into the scenario. Output is written to a file, and each line of the file is on the format "time number_of_neighbors".

- **parse_tcpdump.py:** Crawls the test output directory and parses all nodes' tcpdump files. Computes the number of sent and received messages as well as the resource utilization.

The gnuplot-files for making plots from the output from these scripts are not included.