Context Discovery in Ad-hoc Networks

Fei Liu

Graduation committee:

Chairman:	Prof. dr. ir Anton J. Mouthaan
Promoter:	Prof. dr. ir. Boudewijn R. Haverkort
Assistant promoter:	Dr. ir. Geert Heijenk
Members:	
Prof. dr. Marilia Curado	University of Coimbra, Portugal
Prof. dr. ir. Erik R. Fledderus	Eindhoven University of Technology/TNO
Prof. dr. ir. Sonia Heemstra de Groot	Delft University of Technology/ Twente Institute for Wireless & Mobile Communications
Prof. dr. Hans van den Berg	University of Twente/TNO
Prof. dr. ir. Kees C.H. Slump	University of Twente

CTIT Dissertation Series No.11-200
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217, 7500 AE Enschede

ISSN 1381-3617
ISBN 978-90-365-3207-5
DOI 10.3990/1.9789036532075

Publisher: Wöhrmann Print Service
Cover design: Fei Liu and Wouter Hermelink

CONTEXT DISCOVERY IN AD-HOC NETWORKS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
Prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties,
in het openbaar te verdedigen
op donderdag 9 juni 2011 om 12.45 uur

door

Fei Liu

geboren op 17 september 1980
te Changzhou, China

Dit proefschrift is goedgekeurd door:
Prof. dr. ir. Boudewijn R. Haverkort (promotor)
Dr. ir. Geert Heijenk (assistent-promotor)

Abstract

With the rapid development of wireless technology and portable devices, mobile ad-hoc networks (MANETs) are more and more present in our daily life. Ad-hoc networks are often composed of mobile and battery-supplied devices, like laptops, mobile phones, and PDAs. With no requirement for infrastructure support, MANETs can be used as temporary networks, such as for conference and office environments, and for disaster areas. The disadvantage is that they usually have limited bandwidth and that devices in ad-hoc networks have energy-constrained power supplies, which requires simple and efficient underlying communication protocols. One of the most fundamental actions that such devices in networks need to do is to find information about the environment they are operating in. To share and use the available context information in the network, devices first need to discover and locate the required information. This action is called context discovery. However, none of the existing discovery protocols can well support resource-limited fully-distributed MANETs. Therefore, in this thesis, we design and develop a new context discovery protocol for MANETs, which is called `Ahoy`.

By using attenuated Bloom filters, `Ahoy` reduces traffic load to discover available context information and provides directional probabilistic querying. We build an analytical model to evaluate the performance of `Ahoy` compared with two conventional approaches: pro-active and reactive discovery protocols, and to allow for optimization of `Ahoy`'s parameters. The results of the analytical model are validated by simulations. We estimate the network traffic generated by `Ahoy` in both static and dynamic environments. We find that `Ahoy` requires significantly less network traffic than the other two protocols in static networks, and that it is stable in a dynamic environment in which the network topology may change.

We also study the vulnerability of `Ahoy` when it encounters different malicious attacks. Our analyses shows that compared with pro-active and reactive protocols, `Ahoy` is not more vulnerable than the other two protocols. In some cases, the use of attenuated Bloom filters can even help to protect the contents of packets up to a certain level. In case of serious risks, we propose specialized security countermeasures to enhance the network security of `Ahoy`.

Finally, we build a prototype of `Ahoy` and test it on UNIX-like platforms.

Through these analysis and studies, we conclude that the novel discovery protocol `Ahoy` proposed in this thesis can discover information efficiently, while generating only little network traffic, in both static and dynamic fully-distributed MANETs.

Contents

1 Introduction **1**
- 1.1 Background . . . 2
- 1.2 Motivation . . . 6
- 1.3 Design Requirements and Assumptions . . . 8
- 1.4 Research Questions . . . 9
- 1.5 Approach and Dissertation Structure . . . 11

2 Context and Service Discovery Protocols **15**
- 2.1 Overview . . . 15
  - 2.1.1 Information Description . . . 16
  - 2.1.2 Storage Methods . . . 17
  - 2.1.3 Discovery Methods . . . 18
- 2.2 SDPs for MANETs . . . 19
  - 2.2.1 Centralized approach . . . 19
  - 2.2.2 Cluster-based approach . . . 20
  - 2.2.3 Distributed Approach . . . 21
- 2.3 Discussion . . . 23

3 Context Discovery Using Attenuated Bloom Filters **27**
- 3.1 Bloom Filters . . . 28
  - 3.1.1 Basic Concept of Bloom filters . . . 28
  - 3.1.2 Two Basic Functions . . . 30
  - 3.1.3 Attenuated Bloom Filters . . . 34
- 3.2 Protocol Overview . . . 35
- 3.3 Context Exchange . . . 38
  - 3.3.1 Context Aggregation . . . 38
  - 3.3.2 Context Exchange . . . 40
  - 3.3.3 Design Choices . . . 43
- 3.4 Context Query . . . 48
  - 3.4.1 Context Query . . . 48

3.4.2 Design Choices 49
3.5 Context Update and Maintenance 57
3.6 Discussion 63

4 Performance Modeling **65**
4.1 Modeling Preliminaries 66
4.1.1 Network structure 66
4.1.2 Connectivity in Ad-hoc Network Models 67
4.2 Cost Functions 74
4.2.1 General Assumptions and Related Vital Parameters 75
4.2.2 General Functions 76
4.2.3 False Positive Probability 78
4.2.4 Packet Size 81
4.3 Analysis of two Reference Protocols 83
4.4 Experimental Results 84
4.4.1 Basic experiments 85
4.4.2 Extensive experiments 90
4.4.3 Summary 94
4.5 Model Validation 98
4.5.1 Brief Introduction to Simulation Model 98
4.5.2 Proof of Equivalent Overhead Cost 99
4.5.3 Comparison setup 101
4.5.4 Experimental Results 102
4.5.5 Summary 105

5 Dynamic Connectivity in Mobile Environment **107**
5.1 Probability of Updating 108
5.2 Grid Structure 110
5.2.1 Link Disappearance 110
5.2.2 Link Appearance 112
5.2.3 Node Disappearance 115
5.2.4 Node Appearance 116
5.2.5 One Moving Node 121
5.2.6 Summary 130
5.3 Circular Structure 132
5.3.1 Simulation Setup 132
5.3.2 Node Disappearance 133
5.3.3 Node Appearance 137
5.3.4 Packet Loss 139
5.3.5 One Moving Node 142

5.3.6 Summary . . . 145
5.4 Comparison between three protocols . . . 147

6 Vulnerability Analysis **149**
6.1 Summary of Attacks . . . 150
6.2 Damage from the attacks . . . 152
6.3 Privacy Intrusion . . . 154
6.3.1 Sniffing of advertisement packets . . . 154
6.3.2 Sniffing of query packets . . . 156
6.3.3 Sniffing of reply packets . . . 158
6.3.4 Summary . . . 158
6.4 Lower Discovery Efficiency . . . 159
6.4.1 Modification . . . 159
6.4.2 Packet dropping . . . 168
6.4.3 Replay . . . 169
6.5 Network Jamming . . . 170
6.5.1 Flooding advertisement packets . . . 170
6.5.2 Flooding query packets . . . 171
6.5.3 Flooding reply packets. . . . 173
6.5.4 Summary . . . 173
6.6 Countermeasures . . . 174
6.6.1 Encryption . . . 175
6.6.2 Michael: Message Integrity Code . . . 176
6.6.3 Authentication algorithms . . . 176
6.6.4 Rule management . . . 177
6.7 Summary . . . 183

7 Proof-of-Concept Implementation **185**
7.1 Implementation Choices . . . 186
7.1.1 Context Information Type Format . . . 186
7.1.2 Context Duplication . . . 187
7.1.3 Query Format . . . 187
7.1.4 Query Method . . . 187
7.1.5 Route Recording . . . 188
7.1.6 Means of Query Propagation . . . 188
7.1.7 Underlying Protocols Support . . . 188
7.2 Message Type and Message Format . . . 189
7.2.1 Address . . . 189
7.2.2 Advertisement . . . 190
7.2.3 Query . . . 191

7.2.4 Reply . . . 191
7.2.5 Keep-alive . . . 192
7.2.6 Update request . . . 192
7.3 Functional Description . . . 192
7.3.1 Event and State Variables . . . 192
7.3.2 Initialization . . . 194
7.3.3 `Ahoy` Advertisements . . . 195
7.3.4 `Ahoy` Queries . . . 195
7.3.5 `Ahoy` Responses . . . 196
7.3.6 Keep-Alive Messages . . . 196
7.3.7 Update-Request Messages . . . 197
7.3.8 User Advertisements . . . 197
7.3.9 User Revocations . . . 198
7.3.10 User Queries . . . 198
7.3.11 The Keep-Alive Timer . . . 199
7.3.12 The Advertisement Timer . . . 199
7.3.13 The Query Cache Cleanup Timer . . . 199
7.3.14 The Service List Cleanup Timer . . . 200
7.3.15 Query Timeouts . . . 200
7.3.16 Shutdown . . . 200
7.4 Testing and Results . . . 200
7.4.1 Test Goals and Settings . . . 200
7.4.2 Test scenarios . . . 201
7.4.3 Test results . . . 204
7.5 Discussion . . . 211

8 Conclusions and Future Work 213
8.1 Conclusions . . . 213
8.2 Future Work . . . 216

A Figures of the Overhead Cost by Ahoy, the Proactive and the Reactive Protocols with Different Paramters 219

B The Probability Distribution of the Number of Bits Set 235

Bibliography 238

About the author 247

Acknowledgements 251

Chapter 1

Introduction

Nowadays, more and more people have portable wireless devices, such as laptops, PDAs, and mobile phones. These devices are used in mobile ad-hoc networks (MANETs), in which people can share information and services among each other. One of the essential functions in MANETs is to support context information discovery. Context discovery protocols should be capable to find and locate information that is distributed in the network. These protocols should be simple and efficient, due to the limited bandwidth that is available in MANETs, and due to the limited energy capacity of the battery-powered devices.

Existing discovery protocols cannot fulfill both requirements at the same time. In this thesis, we therefore propose a novel space-efficient context discovery protocol for resource-constrained MANETs. The protocol is named `Ahoy`. It uses Attenuated Bloom Filters (ABFs) to represent context information types in the network. Compared with conventional solutions, such as proactive and reactive protocols, it consumes less storage space for information, supports selective querying, and reduces the traffic generated for discovering information in the network. `Ahoy` thus helps to save bandwidth and transmission power which is essential for ad-hoc networks.

This chapter is organized as follows: Section 1.1 introduces background information. The motivation for the thesis is presented in Section 1.2. Then, we discuss the design requirements and assumptions, in Section 1.3. Thereafter, we pose the main research questions in Section 1.4. Section 1.5 elaborates on the approach and the structure of the thesis.

1.1 Background

Wireless technology is developing rapidly, and in recent years, it has been deployed in many different application areas, from personal devices to satellites. We can connect to almost every device using wireless technology. More and more consumers possess personal devices, such as PDAs, laptops, and cell phones, which are facilitated by wireless communication technology, such as Bluetooth [7] and WIFI [39]. As a result, research and application developments are extending from the traditional wireless access networks to networks with a more direct communication manner: mobile ad-hoc networks (MANETs).

MANETs, unlike Ethernet and infrastructure Wireless LAN (WLAN), do not rely on fixed infrastructures. Devices can establish an arbitrary network via wireless communication when needed. We call the devices that establish the network, the nodes of the network. The wireless communications between nodes are called links. Generally, nodes are not required to load or exchange configuration files to form or join the network [73]. Often nodes are battery-powered and able to move freely in any direction at any speed, which can lead to a frequent variation in connectivity. Wireless technologies that are used in MANETs, like Bluetooth, have improved in recent years. However, they still cannot provide the same data rate and bandwidth as Ethernet and infrastructure WLAN.

MANETs can be used in various situations, especially when no infrastructure support is available or when it is too time-consuming and expensive to set up an infra-structured network. Normally, MANETs are temporary networks and their topology is unpredictable and dynamic. Typical scenarios in which MANETs are established include [71]:

- **Office and conference centers.** In such an environment, most of the resident devices, such as desktop computers, printers, and scanners, are generally connected to Ethernet or WLAN. However, the mobile devices from employees or visitors, are often not authorized to connect to the fixed network. Consider a meeting scenario, where many visitors coming from cooperating partners need to share documents, exchange name cards, and use printers. Ad-hoc networks can satisfy such needs.

- **Disaster relief areas.** One of the major applications for ad-hoc networks

is emergency rescue. It is usually not possible to establish an infrastructure network in areas damaged by nature or man-made disasters, such as fires, explosions, tornados, or earthquakes. Buildings and base stations are badly damaged or destroyed and there is no time to build up a fixed network to facilitate rescue teams. However, rescue teams can build up their own ad-hoc networks to communicate with each other. The refugees can also join the networks and provide crucial personal information like their location and health status, via their personal mobile devices such as cell phone or global positioning system (GPS). Meanwhile, the rescue team can offer first-aid instructions to them.

- **Personal environments.** With the surprisingly fast development of personal devices and their applications, regular consumers can establish their own personal networks, which may contain cell phones, laptops, wireless keyboards and mouses, gaming devices, cameras, and video recorders. Furthermore, people can share devices with friends. This kind of network can be set up at home as well as in restaurants, theaters, cinemas, and even in high-speed moving objects like cars and trains.

- **Remote areas.** Ad-hoc networks can also be set up in remote open areas where it is difficult to build fixed networks. This type of network is commonly used to support research works, like in polar areas, glaciers, high mountains, and forests.

Figure 1.1 visualizes an example of a MANET in an office scenario. Various devices are connected to each other via wireless links. They form an ad-hoc network to provide information and services to each other.

MANETS can appear in specialized forms, such as wireless sensor networks (WSNs) [2] and vehicular ad-hoc networks (VANETs) [46]. WSNs are mostly composed of small devices like sensors. These sensors collect data that are sent to some central servers for further processing. Compared with normal MANETs, the devices in WSNs often have relatively little storage capacity and processing power, and they are generally less mobile. In this respect, VANETs can be considered to be another extreme. Devices in VANETs are equipped in vehicles which move around. Such networks are highly dynamic, as the speeds of the moving devices can be very

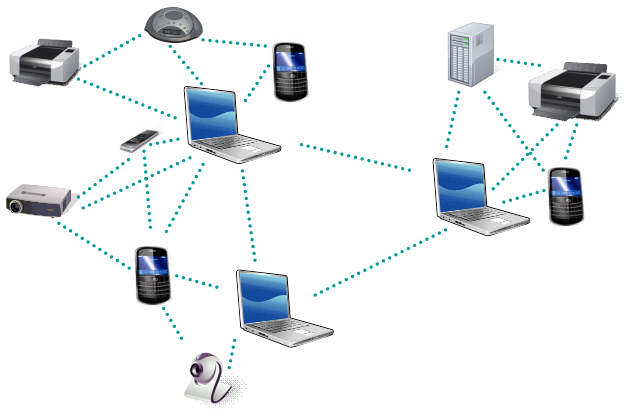


Figure 1.1: A MANET in a conference center, which consists of computers, PDAs, webcams, printers, projectors, and phones.

different. For example, on a highway cars can approach each other quite rapidly, especially in (nearly) congested traffic flows. As a result, two vehicles can move inside and outside each others communication range with high frequency. On the other hand, devices in VANETs can be as powerful as a normal computer. The constraints regarding the small devices in WSNs are therefore not a concern in VANETs. In this thesis, we do not consider extreme forms of MANETs such as WSN and VANETs, but focus on more regular MANETs composed of personal devices, such as PDAs, laptops, and cell phones. These devices are often battery powered. In general, they have less processing power than PCs and servers, but more than sensors. Such devices are carried by people, which may move with high speed, such as in high-speed trains. However, we assume that the relative movements between the nodes in the network are not as large as in VANETs.

For networks without predefined topologies, a major challenge is to find and locate the desired information source that is being requested by an arbitrary device. This action can be defined as *context discovery*. In computer science, the most referred definition of *context* is given by Dey et al. [22] as:

> *Context is any information that can be used to characterize the situation of entities (i.e. whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including*

> *the user and the application themselves. Context is typically the location, identity and state of people, groups and computational and physical objects.*

In this definition, context actually refers to *context information*. For example, if the entity is a printer, its context includes color, location, queue length, etc. In this thesis, we categorize context information into types, called *context information types*. For example, we have the following two contexts: "Device A is located in room B" and "Device A is at the third floor". Both contexts describe the detailed location of Device A. They can therefore be categorized into the same type "location of Device A". In MANETs, every node plays two roles. It can be both a *context source* which provides the context information, and a *context user* which looks for and uses available information. In this thesis, *context discovery* is defined as follows:

> *Context discovery is the action to discover where (the) relevant context information is located.*

When one looks for context information, one generally first queries for the type of the requested information. For example, when we want to know where Device A is, we ask our neighboring nodes to provide us with the "location of Device A". When we find the context source that can give us this context information, we then retrieve the detailed contents.

To make sure that nodes understand each other, it is important to standardize the context information types in the network. We assume that the context information types are standardized by specific names, and that all nodes know these standard names. From now on, when we refer to "context" or "context information", we actually mean the type of context information.

Recently, much effort has been spent to develop protocols for context discovery. However, most of these efforts have been related to networks in which information is centrally stored, and the proposed methods are less suited for decentralized ad-hoc networks. In this thesis, we present a novel discovery protocol for simple and cost-efficient context discovery in ad-hoc networks.

1.2 Motivation

When devices just arrive in a new environment, they do not have any idea about which context information is available around. Before they take any action, i.e., establish communication links with other nodes, they often want to learn first what is available around and whether there is any relevant context information reachable. In this type of networks, which is called *context-aware networks*, an overview of available context information is provided to nodes that would like to join the network. Nodes establish links, based on this knowledge. This concept has also been defined in the Freeband Project AWARENESS (context AWARE mobile NEtworks and ServiceS) [25], which mainly focused on the development of services and network infrastructures for context-aware and proactive applications. The research described in this dissertation has been performed in the context of the Freeband AWARENESS project, and it aimed to study and design a context discovery protocol for context-aware MANETs.

Context discovery in such networks faces some serious challenges. First of all, nodes should be able to share the available context information with other nodes. Moreover, there are challenges which are mostly related to special features of MANETs, according to [67] and [14], as follows:

- **Unstable wireless links.** Nodes connect to each other via wireless links, which are not as reliable as wired connections. The quality of the transmission can be affected by, e.g., weather, temperature, and the surrounding environment.

- **Mobility.** Nodes have the freedom to move. Therefore, links between nodes change frequently, and network topologies vary accordingly. As a consequence, the locations of context information also change frequently.

- **Arbitrary and decentralized topology.** As a result of the dynamic structure of ad-hoc networks, nodes are randomly distributed in space. With no base station coordinating the flow of messages, each node forwards packets to and from the others individually.

- **Battery-powered small devices.** Nodes are often battery powered, which offers the advantage of mobility, but also restrains the power consumption.

- **Limited bandwidth.** Due to the wireless communication, ad-hoc networks have limited bandwidth in general. Large packets and frequent packet exchanges can easily jam the network.

- **Self-organized and self-configured.** Nodes are capable of configuring by themselves with little or no human interference to join the network and reconfigure themselves automatically as the network changes.

We claim that existing discovery protocols cannot handle these characteristics and challenges in a satisfying way.

We can categorize existing discovery protocols by the way they store information. According to this classification, which is described in detail in Section 2.1.2, we can distinguish between the following three types:

- centralized approach;

- cluster-based approach;

- distributed approach.

The centralized approach requires master or gateway nodes to maintain directories. This requires some sort of hierarchy in which there is sufficient storage capacity in the "servers". Dynamic ad-hoc networks mainly consist of mobile nodes, which have low storage capacities. The centralized approach is therefore less suitable for fully distributed ad-hoc networks. With unknown topology and no pre-defined infrastructure, it is also not possible to establish groups or organize clusters in advance. The distributed approach seems to be the only suitable approach for ad-hoc networks. The existing discovery protocols that use the distributed approach are, however, not very efficient. First, information is often advertised and cached in space-consuming formats, such as textual, attribute-value pairs, or markup languages. Second, the question of how much information should be advertised, is so far not resolved. This is a fundamental question, because it determines the efficiency of the context discovery. When for example more information is advertised in advance, nodes know more often where to look for information. As a result, they can query efficiently for information. However, the extra advertisements will

also generate more traffic. We thus need to find a balance in the amount of advertised information and query efficiency to obtain a discovery protocol that meets the required high efficiency.

Therefore, we are looking for a method to support efficient information representation and storage for the discovery phase. We aim to develop a multi-hop discovery protocol for fully distributed context-aware MANETs, which provides nodes an overview of existing context sources, but in the process tries to minimize the amount of generated traffic and required computational power.

1.3 Design Requirements and Assumptions

From the disparity between the characteristics of dynamic ad-hoc networks and existing protocols, the design assignment for the development of a new discovery protocol requires special attention. Those design requirements are addressed below.

- **Context-aware networks.** Nodes that participate in the network should have an overview of available context sources around. The new protocol should provide this information to every node in the network, starting from the moment when it joins the network.

- **Efficient information representation and discovery process.** Ad-hoc networks have limited bandwidth and processing power. Space and traffic savings are the keywords for the discovery protocol design. Packet sizes should be small, and frequent packet exchanges throughout the network should be avoided.

- **Simple computation during the discovery process.** Battery-powered nodes in ad-hoc networks cannot afford heavy computation load. The complexity to update information and search for required information should be small. Even in a high density network with a lot of information updates and frequent discovery requests, the new discovery protocol should limit the power-consumption for nodes.

- **Decentralized approach.** Nodes in MANETs are mobile and mostly battery-power supplied. They might run out of power or move to other places at any

time. We can not rely on centralized discovery approaches where one or few node(s) keep records or directories of all context information in the network. The new design should support discoveries in decentralized topologies, where no node performs as a "server" or a gateway node.

- **Discovery in a mobile environment.** There are many dynamic factors in ad-hoc networks. Mobile nodes and wireless communication cause variation of the location of information in the network. The new design should deal with those dynamic factors.
- **Multi-hop discovery.** The larger the query range, the more information can be found. The new protocol should be capable of locating information multiple hops away from the querying node.
- **Pre-configuration free.** Nodes should not need to install or download configuration information to join the network.

The design of the new discovery protocol will enable users to locate requested information in context-aware ad-hoc networks. We focus on how to **discover** the information. In this thesis, we do not touch the topic of actually obtaining the information. The protocol design is also independent of the underlying communication protocols. It can be resided in the transport layer, e.g., on top of TCP or UDP, in the network layer, e.g., on top of IP, or in the link layer, just above the technology dependent MAC-sublayer. It should be able to serve any wireless communication network protocol, such as Bluetooth, Zigbee, etc. We do not consider the choice of underlying protocol in this thesis. Moreover, we assume that types of context information are standardized. Each context information type is uniquely known by a specific name, and all nodes are aware of the standard. In other words, when a node looks for a type of context information, the other nodes understand what it is looking for.

1.4 Research Questions

The main objective of this thesis is to propose an efficient context discovery protocol for ad-hoc networks. The main research question of this dissertation is:

How can nodes in a context-aware ad-hoc network find and locate requested context information types fast and precisely, with limited bandwidth usage and power consumption?

We should cope with the following research topics to resolve the main research question during the design.

Research Question 1: Protocol design. How to discover context information to fulfill the mentioned design requirements?

RQ 1.1: *Context representation.*

- What is the proper manner to represent context information during the discovery process?
- How should we record the information availability in the network? Which nodes, if any, may keep lists or directories of available information. What is the best choice for our situation?

RQ 1.2: *Discovery method.* How to find information in a fully distributed network? We want to announce and query the information in a manner that does not generate a large amount of traffic. In general, a fast discovery protocol requires the announcement of detailed context information, which consumes a significant amount of bandwidth and battery power. How can we obtain an efficient protocol, which at the same time limits the consumption of bandwidth and battery power?

Research Question 2: Protocol performance. What is the performance of the new protocol, especially, in terms of the following aspects?

RQ 2.1: *Complexity.* What is the complexity of our protocol? It is important that the protocol itself is not too complex. Complex algorithms may consume too much power for computation and transmission.

RQ 2.2: *Scalability.* How does the protocol perform under different network scale, i.e., small, middle, or large networks, with different network densities? Does the protocol have a relatively reasonable performance in high density networks?

RQ 2.3: *Mobility.* How does the protocol react to network dynamics? Can we still locate information when nodes are moving? Is there any relation between network performance and speed or direction of the moving objects? Mobility is one of the important features of MANETs. It is important to make sure that the protocol is stable and functioning well in dynamic networks.

RQ 2.4: *Vulnerability.* What is the vulnerability of the protocol and how can it be improved? How does the protocol react towards various kinds of attacks? Can we improve the protocol such that the users are protected against some or even all of these attacks?

1.5 Approach and Dissertation Structure

To elaborate on the above mentioned research questions, we take the following approach and organize the thesis as follows. Figure 1.2 gives an overview of the outline of the thesis, especially in relation to the research questions.

- *Chapter 1, Introduction.* The current chapter introduces the backgroud and the motivation of the research in this dissertation, and presents the scope of this research.

- *Chapter 2, Context and Service Discovery Protocols.* We first study related work regarding context and service discovery protocols in MANETs. We argue why the existing protocols cannot fulfill the requirements mentioned in Section 1.3 and why there is a need for a new discovery protocol for decentralized MANETs.

- *Chapter 3, Context Discovery Using Attenuated Bloom Filters.* Based on the requirements posed in Section 1.3 and the related work in Chapter 2, we propose a novel context discovery protocol, **Ahoy**. In this chapter, we elaborate on the detailed protocol design and discuss our design choices. In doing so, we answer Research Question 1.

- *Chapter 4, Performance Modeling.* An analytical model of `Ahoy` is established for a static network. We use the model to optimize the parameter setting of the system, and we evaluate the performance of `Ahoy` by comparing it with that of conventional approaches (so-called proactive and reactive discovery protocols). Finally, we validate the analytical model with simulations. Chapter 4 therefore answers Research Question 2.1 and 2.2.

- *Chapter 5, Dynamic Connectivity in Mobile Environment.* We extend our analysis to the extra network traffic that is generated in mobile networks. We observe different scenarios of dynamic connectivity and their influence on the performance of `Ahoy`. We use both analytical and simulation approaches. Again, the performance of `Ahoy` is compared with proactive and reactive protocols. Chapter 5 answers Research Question 2.3.

- *Chapter 6, Vulnerability Analysis.* In this chapter, we analyze the vulnerability of `Ahoy`. We study and compare how various attacks affect `Ahoy`, the proactive and reactive protocols. Accordingly, we propose countermeasures to avoid such attacks, or alleviate their impact. Chapter 6 answers Research Question 2.4.

- *Chapter 7, Proof of Concept Implementation.* Subsequently, we implement a prototype for `Ahoy`. In this chapter, we elaborate on the implementation details. We test the prototype to verify whether `Ahoy` performs correctly and analyze the amount of `Ahoy` traffic as portion of the total amount of traffic.

- *Chapter 8, Conclusions and Future Work.* Finally, we conclude the thesis and propose future work.

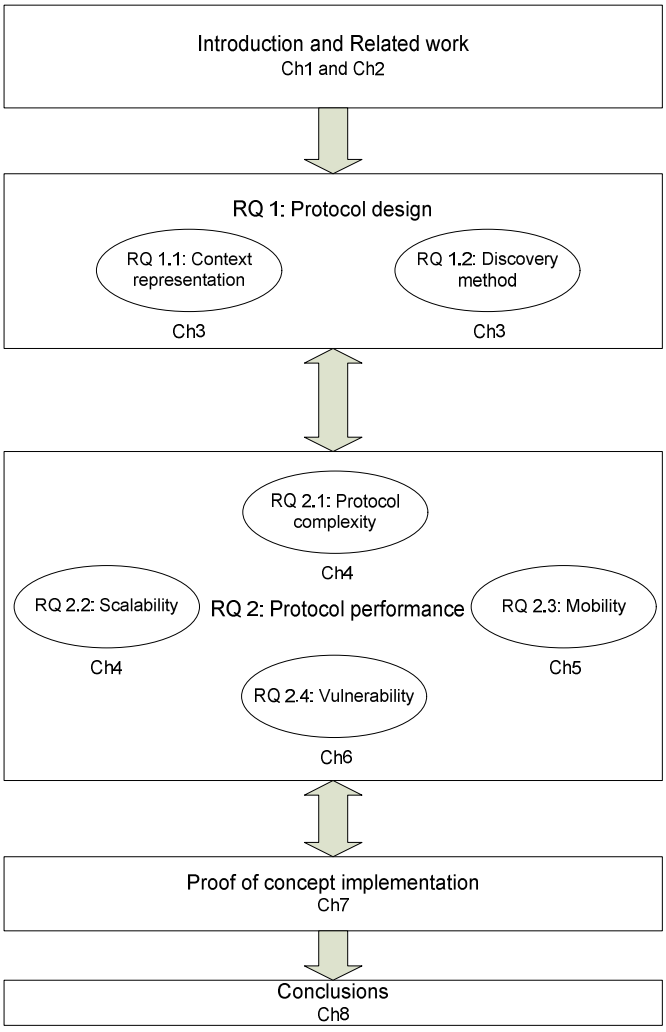


Figure 1.2: The outline of the thesis.

Chapter 2

Context and Service Discovery Protocols

The goal of this dissertation is to design and develop a context discovery protocol for MANETs, so that devices have an overview of existing context information types around and find requested context information quickly and efficiently in a decentralized ad-hoc network. In the literature, a lot of attention has been given to service discovery rather than context discovery. In this chapter, we describe the relation between service discovery protocols and context discovery protocols. Furthermore, we explore existing discovery protocols and show why they are not suitable for our purpose.

This chapter is organized as follows. We first address the relation between context and service discovery and give an overview of context/service discovery protocols in Section 2.1. Then we introduce briefly existing service discovery protocols for MANETs in Section 2.2. Finally, we discuss why existing service discovery protocols can not fulfill our requirements in Section 2.3.

2.1 Overview

In computer science, *context* refers to the circumstances under which a device is being used as defined in Section 1.1, whereas a *service* can be any application (consisting of software and/or hardware) that a user might want to use. *Service discovery* is the action to find requested services. Although the concept of context and service

is different, the methodology to discover them is in essence the same. First, both context and service can be described in a certain format or template. Secondly, the discovery of both can be understood as an action of looking for information in the network. The methods used for service discovery are suitable for context discovery as well. In this chapter, we refer to the generic aspects of the discovery process when using the terms service discovery and context discovery.

Context/service discovery protocols encounter the following three fundamental questions in general:

- How is the context/service information represented?
- Where is the information stored?
- How is the information discovered?

We address these three questions in detail in the remainder of this section.

2.1.1 Information Description

A service may have some *attributes*, which are the characteristics of the service. For example, a printer service can have the following attributes: position, resolution, color, etc. Each attribute can be associated with a *value*, e.g., the *position* of the printer is *Room 101*. Similarly, a context can also be characterized with *context information types*, as we introduced in Section 1.1.

Context/service information can be described into various ways. According to [56], service information and its attributes can be categorized into the following five types:

- *Textual.*
- *Attribute-value pairs.*
- *Hierarchy of attribute-value pairs.*
- *Markup languages.*
- *Object-oriented interface.*

Those forms can define the context/service information in different levels of detail. For example, a context/service can be described with its name in a text string, or with detailed definition of attributes in Markup languages. Detailed descriptions normally require larger storage space, which may also consume more processing effort and communication bandwidth. The choice of description form depends on the requirements of the applications.

2.1.2 Storage Methods

The existing service discovery protocols (SDPs) utilize the following approaches to store information: *centralized approach*, *cluster-based approach*, and *distributed approach*.

- Centralized approach: One or several nodes act as service repositories and store information about the available services or the directory of services. Nodes register their services in the service repositories and query them to retrieve the required information.

- Cluster-based approach: Nodes in the network are grouped into clusters based on certain policies, such as physical location or services they provide. In a cluster, information can be stored centrally in one node or decentralized in many nodes, depending on the protocol design. Intra-cluster discovery is supported by various approaches in different protocols, such as using gateway nodes, or anycasting to other clusters. We address this in detail in Section 2.2.2.

- Distributed approach: Services are stored everywhere in the network. Nodes can either advertise and cache the stored information in advance or not advertise at all, depending on the protocol design. Nodes that need a service broadcast or multi-cast queries to look for the required service.

In the rest of the chapter, we discuss existing SDPs that can be used for MANETs. These are categorized by the way they store service information.

2.1.3 Discovery Methods

Basically, there are three approaches to discover information: the *proactive* approach, the *reactive* approach, and the *hybrid* approach.

- Proactive approach: Nodes advertise local services periodically. When one node receives an advertisement, it stores the services into a table or the location of the services into a directory. When a node requests a certain service, it checks the cached table or directory in cache to locate the service.

- Reactive approach: Nodes do not send any advertisement to announce their service information. Therefore, no node knows where services are available. They simply broadcast queries to all the nodes in the network, whenever they look for a service.

- Hybrid approach: The hybrid approach tries to strike a balance between the proactive and the reactive approaches. Generally, advertisements are sent to a subset of nodes and/or with a limited frequency. This can help nodes to locate required services and reduce the traffic for querying.

The proactive and reactive approaches are conventional discovery approaches, which have the drawback that they flood packets over the network [56]. In proactive protocols, nodes need to send frequent advertisements throughout the network to keep the cached services up to date. This is especially the case when services (nodes) are mobile and change locations all the time. As a result, the proactive approach is not suitable for highly dynamic networks. In contrast, in the reactive approach, nodes flood the network with queries. The amount of traffic is in this case highly related to the query frequency. Often, the reactive approach is used in dynamic networks where frequent changes in network topology makes caching the location of existing services inefficient and costly. The choice of reactive versus proactive approach, to a large extent, depends on the network structure, the query rate, and on the interaction with the underlying routing protocol [37].

Due to the drawbacks of the two conventional approaches, many authors have proposed a hybrid approach, to balance between the traffic due to advertisements and queries, and to avoid the flooding of messages throughout the network. The approaches and policies that are used for this purpose, are different in various discovery

protocols. We address this in detail when we introduce existing service discovery protocols for MANETs below.

2.2 SDPs for MANETs

In this section, we group the existing service discovery protocols into three categories based on how services are stored in the network introduced in Section 2.1.2 and briefly introduce how they work.

Particularly, we study only the existing SDPs for MANETs, because in this thesis we focus on discovery in MANETs where the network topology is neither fixed nor stable. Prominent SDPs, such as Jini [74], Splendor [85], Salutation [77], UPnP [78], JXTA [58], Service Location Protocol (SLP) [30], are designed for enterpriser networks or wide-area networks, which rely on fixed or stable network structures. Therefore, these discovery approaches cannot be applied for our purpose and are out of our interest.

2.2.1 Centralized approach

Bluetooth Service Discovery Protocol [7] defines a service record as the entire list of attributes of the provided services. This record is stored in an SDP server. Clients can obtain the service information via SDP servers. A node can act both as a SDP server and as a client, depending on whether it provides or requests services.

In CDS [26], services are described by attribute value pairs and registered in a set of nodes called Rendezvous Points (RPs). A hash function is applied to obtain each attribute-value pair, and the hashed results are registered. Queries are only directed to the relevant RPs. A load balancing matrix of RPs is used to avoid flooding queries to one node.

Kozat and Tassiulas proposed a directory-based discovery mechanism DSDP [47] using backbone structures. A set of relatively stable nodes are selected to form a dominating set. Based on this dominating set, a mesh network with a virtual backbone is constructed. Nodes in the dominating set act as directory agents (DAs) and process services requests from nodes.

FRODO (originally named as SDP@HA) [75] is a service discovery protocol designed for the home environment. It uses a typical centralized architecture, where one node in the network is elected as "central". This node stores the service Lookup table, and other nodes query the "central" whenever they look for any information. Another node is assigned as "backup", which will take over the responsibility of the "central" if it fails.

Varshavsky [80] coupled the discovery protocol with the underlying routing protocols (DSR and DSDV) and so developed a cross-layer discovery protocol. Two fundamental components are defined to facilitate discovery: a service discovery library (SDL) and a routing layer driver (RLD). Known servers are stored in a service table in the SDL. Clients look for related servers by checking the SDL. The SDL instructs the RLD to disseminate discovery requests for specific services and routes, and to periodically disseminate advertisements for specific services.

The centralized approach can act as either a proactive protocol or a reactive protocol, depending on whether the servers announce their existence in advance. If servers broadcast their existence frequently, and the nodes know where to find the information, it can be considered a proactive protocol. Otherwise, nodes need to look for the servers every time when they request information. In that case, it is a reactive protocol.

Similar to the protocols that maintain central directories, such as Jini, Salutation, etc., these protocols allow networks to store their information storage in few "servers". The network structure, thus, still needs some form of hierarchy in which the server nodes are more important than the nodes that are merely clients.

2.2.2 Cluster-based approach

In Intentional Naming System (INS) [1], services are defined in hierarchical attribute-value pairs. Nodes are grouped into clusters, where nodes in the same cluster are aware of all information from each other. Usually a service directory is used to support inter-cluster queries.

The service ring protocol [45] is a typical clustered hierarchical approach. Nodes are grouped into rings, if they are physically close to each other and offer similar

services. Each service ring has a Service Access Point (SAP), which stores information of the ring. Higher-level rings can be constructed by SAPs, with higher-level SAPs to store services they provide. Nodes can query SAPs for intra- and inter-ring service discovery.

The LANES protocol [45] groups nodes into lanes. Nodes in the same lane broadcast their services and cache them. Whenever a query to certain services cannot be found in the same lane, the query is anycast to other lanes.

Generally, cluster-based protocols proactively maintain routing and service information inside a zone, while using a reactive search approach between the zones. They can be considered as hybrid service discovery approaches, in which clear information storage structures need to be defined in advance.

2.2.3 Distributed Approach

DEAPspace [62] supports single-hop discovery in short-range wireless systems. Each node keeps a list of all services in the network. The information is spread when a node broadcasts its services, and the list of other known services, to the neighbors. The neighbors use this information to enlarge their lists, and they broadcast it to their neighbors. In this way, all the information is distributed through the network.

The Group-based Service Discovery protocol (GSD) [12] is a distributed service discovery protocol for MANETs. Services are described based on DARPA Agent Markup Language (DAML+OIL). Advertisements are sent periodically to nodes within a maximum number of hops. Each node keeps a list of local and remote services that a node has received from advertisements. Services are also grouped to ease discoveries by selectively forwarding queries.

Helmy [36] proposed a zone-based resource discovery protocol. Every node has a "zone", which include all the nodes that are less than a certain number of hops away. It maintains available resource information and routes to all nodes in the zone. It also knows several contact nodes outside the zone. Via the contact nodes, the node is able to discover resources outside its zone.

Lenders et al. [49] proposed fully distributed service discovery protocols. Service instances periodically broadcast advertisements containing their services within a certain scope. Nodes receive the broadcast, cache the information for limited time,

and aggregate advertisements into one single packet and propagate it. Meanwhile, a "potential" is assigned to the cached information, based on the distance to the service provider. When a node looks for certain services, it checks the cache and forwards the query only to the neighbors with the highest "potential".

Allia [68] uses peer-to-peer caching of services between nodes. In Allia, every node broadcasts its local services to nodes in its vicinity, and caches received broadcasted services from neighboring nodes. Allia defines the concept of "alliance" of a node as a set of nodes which local services are cached by that node. When a node queries for a certain service, it looks at its local service list and its local cache. If no match has been found, it checks the caches of members of its "alliance". Nodes that receive the query decide to process it or drop it, based on a predefined policy.

Frank and Karl [24] proposed a cross-layer discovery protocol, which is bounded by the underlying routing protocol AODV. Nodes announce local services within a certain scope and cache the ones they receive. Negative service announcements are used to remove cache entries in corresponding nodes. When a node queries a certain service, a routing packet with the description of the queried service is created. The packet is propagated as a normal AODV routing packet. If a queried node knows the matching service provider, it fills in the destination address into the packet.

The adaptive service discovery model [59] uses a combination of a proactive and a reactive protocol to avoid flooding of advertisements or queries in the network. Nodes control the ratio between the bandwidths used for advertisements and queries. This is done by regulating the frequencies of advertisements and queries. Nodes observe the frequency of advertisements and queries in the network, and based on these observations, determine whether or not to send an advertisement or a query.

Konark [35] is a service discovery and delivery protocol designed specifically for ad-hoc, peer-to-peer networks. Services are described in XML. Each node maintains a tree-based structured service registry in its own SDP manager to store local services, and services that are discovered or received via advertisements within a certain lease time. Any node can query fixed groups of nodes for information. In response to queries, a node which contains the required information advertises (part of) its registry. The services in the received advertisements can also be cached into a local registry. This protocol is a combination of a proactive and a reactive protocol. Nodes do not send advertisements actively, unless there is a match with a query.

Discovered information is cached in nodes for future use.

HESED [83] is a service discovery protocol based on multicast query and reply. After a client multicasts its query, all matching service providers multicast their information to all nodes, which in turn cache this service information. Clients evaluate and utilize the cached information, thereby reducing the number of queries. HESED also eliminates the effect of asymmetric links, providing reliable connections that can be utilized by the forwarding algorithms. However, intermediate nodes do not send replies even if they have some knowledge that is related to the query.

DEAPspace is a typical proactive protocol. Konark and HESED are two reactive protocols which do not actively advertise local services to other nodes. In both protocols, queries are sent within a certain range, and requested information is cached for future use. These protocols thus enhance query efficiency and reduce query traffic, but they require sufficient storage space for nodes to cache the information.

The other protocols use a hybrid discovery approach, which contains both advertisements and queries. Basically, they use two approaches to avoid large amounts of advertisements and queries. In the adaptive service discovery model [59], the advertisement and query frequency is regulated. The more popular approach, however, is to advertise service information within a certain range and to cache the information in the advertisements. Nodes can easily locate existing services within a certain range without querying the entire network, but like the reactive protocols, Konark and HESED, they require that nodes have enough storage space and broadcast extra advertisements.

2.3 Discussion

We summarize the discussed protocols in Table 2.1.

Compared with the distributed approach, the centralized approach and the cluster-based approach are more centralized, because they use servers, service access nodes, etc. In ad-hoc networks, nodes are often mobile and battery-powered. There are frequent changes of network topology and limited bandwidth, and nodes have limited computational and storage capacities. Given these characteristics, we suggest that we can only obtain a robust discovery protocol when the network has no hierarchical structure. This implies that information should be distributed in a

decentralized way, and that, in principle, each node is equally important. In this way, there is little danger that nodes are overloaded, or that the discovery protocol will break down when nodes are disconnected from the network.

We therefore suggest that the distributed approach is the only suitable approach for dynamic ad-hoc networks. The three discovery approaches, *proactive*, *reactive*, and *hybrid*, can all be applied in the distributed approach. The drawback of fully distributed networks is that the information storage is not very efficient. This means that relatively much traffic will be generated in the discovery process. Clearly, the proactive and reactive approach have the greatest risk of flooding the network with packets. Hybrid protocols are probably most efficient in reducing traffic. In that sense, regulating advertisement and query frequencies, such as in the adaptive service discovery protocol, could be a good solution for a static network where there are few requests for services. However, when network topologies change frequently and/or there is a high demand for services in the network, such a discovery protocol may be less efficient.

The caching of existing services is also a good way to reduce query traffic. It is is used in many protocols (GSD, Allia, Konark, HESED, etc.). Caching available information within a limited range is also in line with our requirement of context-aware networks in Section 1.3. Newly arrived nodes need to obtain an overview of available context sources which surround them. To exchange and cache available context sources is a good manner to facilitate nodes with the knowledge of available context sources which is also the purpose of context-aware networks. Moreover, the amount of exchanging and caching can be constrained to a limited range (i.e. certain number of hops), as nodes only need to be aware of context sources in the direct environment.

There are three key questions that need to be taken care of in this respect:

1. How often, how far, and how to advertise services/context?
2. How to cache services/context?
3. Which level of detail of the service/context information is needed during the discovery phase?

The existing protocols focus on resolving the first two questions. However, we believe that the third question may be the most fundamental one. The service

description format is directly related to the size of the advertisement packets and to the storage space used for caching. This is crucial, since the nodes in our ad-hoc networks have limited bandwidth and storage space. Services are currently often represented in textual, attribute-value pairs, hierarchy of attribute-value, Markup languages, or object-oriented interfaces [56], which can contain a lot of information, but also consume large storage space. However, not all this information is necessary for service discovery. For example, suppose that a node looks for the "temperature in Room A". It is only interested in finding a node which provides the service of "temperature in Room A", but not in the detailed attributes of the service, such as "age", "position", and "brand" of the thermostat. It is possible that the node will request information about the "age" in the future. If the level of detail of the advertised context is limited, it may have to send extra queries in the future.

In short, to save network traffic in service/context discovery, it is essential to study how services or contexts can be represented in a space-efficient manner, but with enough detail to limit the amount of queries during the discovery phase.

In this dissertation, we aim to study and find such an information representation method, which enables us to save storage space and network traffic during the discovery process. Using this information representation method, we will design and develop a new context discovery protocol for context-aware MANETs. As we argued above, the new protocol will use a hybrid distributed approach in which nodes advertise and cache available context information types within a limited range.

Table 2.1: SDPs for MANETs.

Protocols	Description	Storage method	Discovery method
Bluetooth SDP [7]	attribute-value	centralized	reactive
CDS [26]	attribute-value	centralized	proactive
DSDP [47]	-	centralized	reactive
FRODO [75]	-	centralized	proactive
Varshavsky et al. [80]	-	centralized	hybrid
INS [1]	hierarchical attribute-value	cluster-based	hybrid
The service ring protocol [45]	-	cluster-based	hybrid
Lanes [45]	-	cluster-based	hybrid
DEAPspace [62]	attribute-value	Distributed	proactive
GSD [12]	Markup Language	Distributed	hybrid
Helmy [36]	-	Distributed	hybrid
Lenders et al. [49]	-	Distributed	hybrid
Allia [68]	-	Distributed	hybrid
Frank and Karl [24]	-	Distributed	hybrid
Adaptive service discovery protocol [59]	-	Distributed	hybrid
Konark [35]	Markup Language	Distributed	reactive
HESED [83]	-	Distributed	reactive

Chapter 3

Context Discovery Using Attenuated Bloom Filters

Ad-hoc networks are distributed wireless networks in which most nodes are mobile, and have limited power supply. When a node searches for some context information, that information can be available locally or in other nodes that are one or multiple hops away. Local information discovery does not consume network bandwidth, and is therefore not considered in this thesis. We focus on multi-hop context discovery in ad-hoc networks. The most important question is how to find and locate the required information. Announcing context information, querying, and determining the location of the context source might generate a lot of traffic. In a high-density network, such traffic can be rather heavy. As a result, the nodes consume quite an amount of power and bandwidth for querying. An efficient context discovery mechanism needs to be developed for such situations.

In this chapter, we propose a novel approach to discover context information in ad-hoc networks, which is named `Ahoy`. `Ahoy` is a decentralized space-efficient discovery method, which reduces network traffic during the discovery process. It represents context information into attenuated Bloom filters for advertising, and supports a directional query mechanism.

There are three phases in `Ahoy`: context exchange, context query, and maintenance and update. In Section 3.1, we first introduce the concept of Bloom filters. In Section 3.2, we briefly introduce the `Ahoy` protocol. In Section 3.3, 3.4, and 3.5, we describe the three phases of the protocol in detail, and we summarize in Section 3.6.

This part of the work has been published in [50].

3.1 Bloom Filters

According to the discussion of Section 2.3, the first phase of context discovery is to locate the nodes which provide the context information that we are looking for. For reducing traffic and facilitating context-aware networks, it is a good idea to advertise some available information a priori. But it is not necessary to advertise all the details. Instead, during the discovery phase, nodes are only interested in what types of context information are available in their environment. They can retrieve the details later when necessary. For this purpose, we need an efficient way to represent context information types and support a traffic-saving context discovery protocol. Exchanged and cached context information may be compressed to a smaller size, but when a node queries for a certain context information type, it should still be able to learn from the cached information whether or not the information exists.

To achieve this, we propose to use Bloom filters (BFs) to represent context information types. Bloom filters [6] have been proposed in the 1970s to represent sets of information in a simple and space-efficient way and to test which information belongs to the set. They are suitable for compressing information without losing much detail. A Bloom filter can aggregate a set of context types into a bit array, and it can provide the existence of information with high confidence. There is a chance of false positives, but not of false negatives. Information can be easily inserted into a BF, but is difficult to be removed from it. In the remainder of this section, we introduce the concept of Bloom filters.

3.1.1 Basic Concept of Bloom filters

Bloom filters are used to present a set of elements. A Bloom filter B is a bit array of w bits, where the individual bits will be denoted as B_i $(1 \leq i \leq w)$. For an empty set, all bits in the Bloom filter are set to 0:

$$B_i(\emptyset) = 0 \qquad (1 \leq i \leq w). \tag{3.1}$$

There are b independent hash functions, H_j $(1 \leq j \leq b)$, which are used to code the elements. The results of the hash functions are over a range $\{1, \ldots, w\}$. The bit positions which are corresponding to hash results are set to 1. So, the Bloom filter

of an element s can be represented as follows:

$$B_i(s) = \begin{cases} 1, & \exists_j H_j(s) = i \ (1 \leq j \leq b), \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

Two basic operations on Bloom filters are *union* and *intersection*. Operation *union* actually combines multiple sets into one single set. It can be simply implemented by a **bitwise OR** of the corresponding Bloom filters of those sets. The outcome is an aggregated Bloom filter representing a union of multiple sets. Operation *intersection* obtains the common elements from multiple sets. It can be implemented by a **bitwise AND** of all corresponding filters of the sets. The outcomes is a filter representing an intersection of multiple sets.

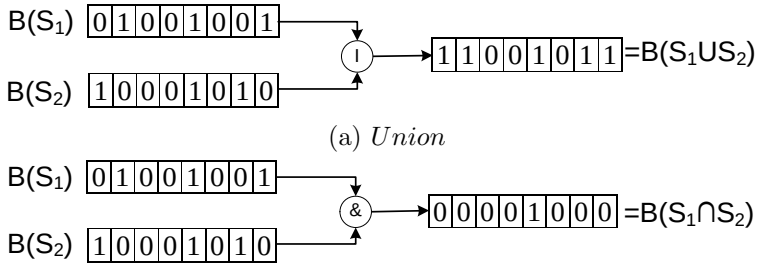
For any two set S_1 and S_2 and their corresponding Bloom filters $B(S_1)$ and $B(S_2)$, the *union* operation can be denoted as:

$$B(S_1 \bigcup S_2) = B(S_1)|B(S_2). \tag{3.3}$$

Likewise, the *intersection* operation can be represented as:

$$B(S_1 \bigcap S_2) = B(S_1)\&B(S_2). \tag{3.4}$$

Figure 3.1(a) and Figure 3.1(b) show examples of the *union* and *intersection* operations respectively.



(a) *Union*

(b) *Intersection*

Figure 3.1: An example of union and intersection.

These two operations are very practical and essential for Bloom filters. Operation *union* can be used to add elements to a filter or to gather information from multiple sources into one filter. Operation *intersection* can check whether information in two filters (for example a query and stored information) matches with each other. We will address these operations further in the next session.

3.1.2 Two Basic Functions

Bloom filters support two fundamental functions: *insert* and *query*. Note that a remove function is not defined for Bloom filters. Below we will introduce the *insert* and *query* functions.

Insert

To insert one element s into a filter, the element first needs to be hashed by b hash functions. Each hash function returns a value, which is associated with a bit position in the filter. This corresponding bit position is set to 1. Inserting is the action of (3.2), continued with the *union* function (3.3). Table 3.1 presents the pseudo code for the above mentioned process.

Table 3.1: Pseudo-code for **inserting** a context information type "s" into a Bloom filter.

```
1  Insert_BF (s){                    % insert "s" into Bloom filter
2      for j = 1 to b {              % apply b hash functions
3          i = H_j(s);               % obtain hash result i
4          B_i = 1;                  % set bit position i to 1
5      }
6  }
```

Query

To query the presence of one element s is to examine whether s is an element of the set S. This function can be performed by the operation *intersection*. The element s is again hashed by b hash functions, in accordance with (3.2). The hash results are b bit positions which are set in a filter $B(s)$. Then, we check the intersection of the two filters $B(s)$ and $B(S)$. If the intersection equals $B(s)$, the element s belongs to the set S. Otherwise, s is considered not to be available in the set. This can be expressed with the following equation:

$$s \in S = \begin{cases} true, & B\left(S\right) \& B\left(s\right) = B\left(s\right), \\ false, & \text{otherwise.} \end{cases} \tag{3.5}$$

Table 3.2 presents the pseudo-code for querying a context information type "s".

Table 3.2: Pseudo-code for **querying** a context information type "s".

```
1   Query_BF (s){                          % Query "s"
2       for j = 1 to b {                   % apply b hash functions
3           i = H_j(s);                    % obtain hash result i
4           if B_i == 0 {                  % if any position is 0
5               return(Not_Exist(s));      % return that "s" does not exist
6               break;                     % stop the whole querying process
7           }
8       }
9       return(Exist(s));                  % return that "s" exist
10  }
```

Example

Let us assume a 6-bit Bloom filter ($w = 6$), using two different hash functions H_1 and H_2 ($b = 2$). There are two information types "*location*" and "*temperature*" available, which need to be encoded with the filter. Suppose, we apply H_1 and H_2 over the information types and obtain:

$$\begin{aligned} H_1(\text{“}location\text{”}) &= 2; \\ H_2(\text{“}location\text{”}) &= 4; \\ H_1(\text{“}temperature\text{”}) &= 3; \\ H_2(\text{“}temperature\text{”}) &= 6. \end{aligned}$$

Therefore, the Bloom filters $B(\text{“}location\text{”})$ and $B(\text{“}temperature\text{”})$ can be represented as the filters in Figure 3.2 and Figure 3.3, respectively. The union of two filters is shown in Figure 3.4. When a query for "*location*" or "*temperature*" is generated, the filter will give a positive answer to it.

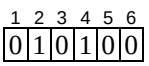


Figure 3.2: The Bloom filter $B(\text{“}location\text{”})$.

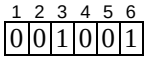


Figure 3.3: The Bloom filter $B($"*temperature*"$)$.

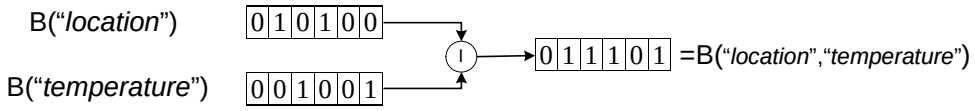


Figure 3.4: The Bloom filter contains both context information types"*location*" and "*temperature*".

False Positives

When several elements are inserted into one filter, multiple bits are set to 1. The combination of those bits can represent not only the available elements, but also non-existing elements. When a node queries a Bloom filter to check the presence of such unavailable elements, the filter returns a positive reply which is not correct. We call such an answer a **false positive** answer [8].

Let us, for example, assume that "*presence*" and "*humidity*" information can be hashed into:

$$H_1(\text{"}presence\text{"}) = 1;$$
$$H_2(\text{"}presence\text{"}) = 4;$$
$$H_1(\text{"}humidity\text{"}) = 2;$$
$$H_2(\text{"}humidity\text{"}) = 6.$$

The filter in Figure 3.4 definitely does not contain "*presence*" information ($\{1,4\}$), as B_1 equals 0, as is shown in Figure 3.5. However, when a node queries for "*humidity*", it returns a positive reply, because the bits B_2 and B_6 are set to 1, as shown in Figure 3.6. In the example, these bits are set by "*location*" and "*temperature*", while "*humidity*" is actually not available. When we query "*humidity*" to the filter, it gives a false positive answer.

When there are more information types encoded in a single filter, the probability of false positives becomes higher. This reduces the accuracy of the query results. However, we can reduce the false positive probability by using a larger filter, which at the same time requires larger storage space. We will address the false positive

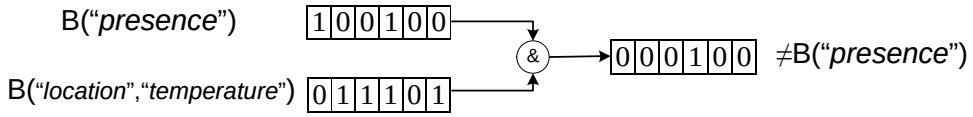


Figure 3.5: When query "*presence*" information, the filter gives a negative answer.

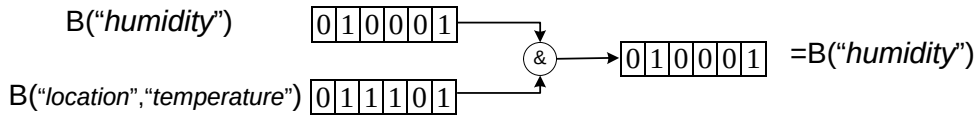


Figure 3.6: When query "*humidity*" information, the filter gives a false positive answer.

probability and its relation with the size of the Bloom filter in detail in Section 4.2.3. Note that Bloom filters do not give false negatives. When the query result is negative, the requested element is sure not belong to the set.

Applications of Bloom filters

Bloom filters were originally designed for hyphenating words in dictionaries [6]. As a space-saving data structure, they are nowadays used to enhance membership queries for sets of information, such as for spelling checks and indexing. A famous application is Google BigTable [13]. Bloom filters have been used to reduce database lookups for differential files [29, 60], and have also been used in various network applications, such as distributed caching, P2P/overlay networks, resource routing, packet routing, and measurement infrastructure [8]. Recently, researchers have explored the application of Bloom filters to ad-hoc networks, for speeding-up cache lookups [64], group management [55], hotspot-based trace back [3], and neighbor solicitation [61].

Service discovery is one of the applications of Bloom filters [8]. Bloom filters are also used as an efficient approach for lossy aggregation and query routing for a Secure Service Discovery Service in [18], where services are locally stored with Bloom filters to speed up queries.

3.1.3 Attenuated Bloom Filters

More recently, Rhea and Kubiatowicz introduced the concept of attenuated Bloom filters (ABFs) in [69] to enhance searching information in peer-to-peer networks. An ABF consists of layers of single Bloom filters. They can be used to provide probabilistic location and routing to enhance querying.

Here, we apply the idea of ABFs to represent context information types for different hop-distances. The number of layers is defined as d. The width of the filter is again denoted by w. From top to bottom, the filter represents the presence of information from close by to further away. In contrast to [69], we define that the first layer (layer 0) of the filter contains the context type information for the current node, while the second layer (layer 1) contains the information of all nodes one hop away. Layer i of an ABF ($0 \leq i \leq d-1$) aggregates all information about context types within i hops away, where layer i is also called the $(i+1)$th layer. The depth of the ABF, d, also stands for the total propagation range of the advertised information. Figure 3.7 gives an example of an ABF of a node, with $d = 3$ and $w = 6$, where the node contains the information "*humidity*" locally, and can reach the information "*humidity*" and "*temperature*" within one hop, and can reach the information "*humidity*", "*temperature*" and "*presence*" in two hops.

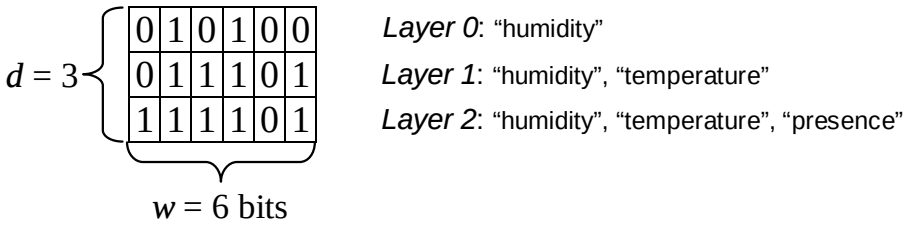


Figure 3.7: A basic 3-layer 6-bits attenuated Bloom filter (ABF) of which layer 0 contains the information "*humidity*", layer 1 contains information "*humidity*" and "*temperature*", and layer 3 contains the information "*humidity*", "*temperature*" and "*presence*".

Attenuated Bloom filters (ABFs) are space saving data structures to store information. They represent information into a limited number of layers of bit arrays. Information is grouped in layers based on a notion of distance (number of hops, in our case). The query of certain information can be done quickly by an *intersection*

operation.

The accuracy of context representation decreases with the number of hops. That is because a node usually can reach more nodes when the number of hops increases. Therefore, more information is stored in the deeper layers, which results in filters with more 1s' set. As a result, there is a higher false positive probability which decreases the location accuracy of elements. For example, layer 0 of the ABF presented in Figure 3.7 contains only one element and layer 2 contains three elements. As a result, there are more 1s' set in layer 2 than in layer 0 and the probability to have a false positive is higher in layer 2 than in layer 0.

The features of ABFs are well suitable for supporting a discovery mechanism. Based on this idea, we propose a fully distributed and lightweight context discovery protocol using attenuated Bloom filters for ad-hoc networks, named `Ahoy`, below.

3.2 Protocol Overview

Generally, there are three phases in context discovery:

- **Context exchange:** nodes **announce** the information regarding to their context information types to the other nodes, when a new network is established or new nodes join an existing network.
- **Context query:** nodes **request** other nodes for information and try to locate the required information.
- **Context maintenance and updates:** nodes keep on **updating** the latest information regarding to their context types to the others.

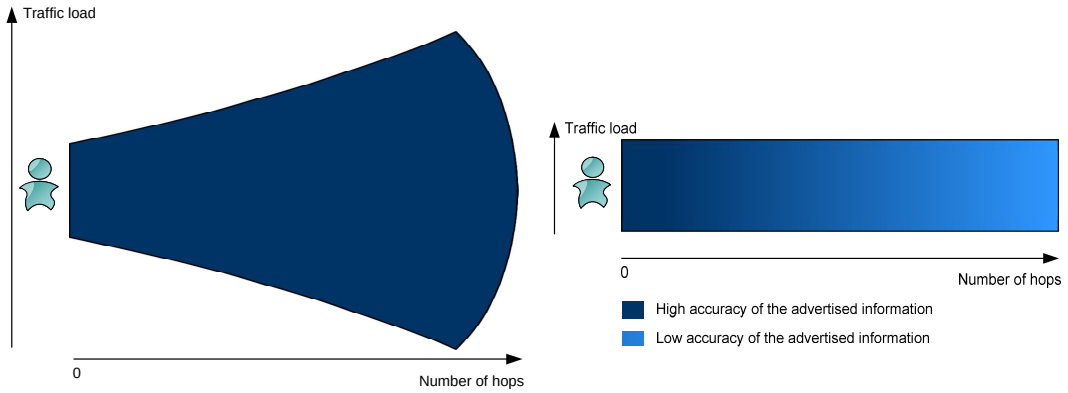
The above three phases are essential, but not compulsory. A discovery protocol can consist of one or more of these phases, as long as nodes can find what they need. For instance, as introduced in Section 2.1.3, the traditional proactive protocol has all three phases. All nodes broadcast to the other nodes what context information they possess. When one node queries information, it directly sends the query to the node which has the context information type. Nodes need to update their information to keep the others informed about the latest information. In a

contrast, traditional reactive protocol only has the context query phase. There is no information exchanged a priori and also no update afterwards. Nodes only query (part of) the network whenever they need information.

The proactive and reactive discovery protocols have both advantages and drawbacks. In the proactive protocol, nodes know where to find which information. There is no extra network traffic consumed to look for the location of the required information. However, this can only be achieved when information is exchanged beforehand, and is constantly kept up to date. This may generate large amounts of network traffic, and it requires extra memory to store the exchanged information, especially, in dynamic networks where frequent updates are required. The proactive protocol is therefore mainly suitable for a static network where information is queried frequently. On the other hand, in the reactive protocol, nodes have no idea about the information availability nor the distribution of information in the network. Nodes need to flood queries in order to find requested information. However, information in the network does not have to be updated. Therefore, this approach is suitable for a highly mobile network in which information is queried only incidently.

`Ahoy` is a discovery protocol which tries to strike a compromise between the proactive and reactive protocols and aims to save network traffic, node memory, and computational power in mobile ad-hoc networks. `Ahoy` contains all three phases: context exchange, context query, and context update and maintenance. It pre-broadcasts some available information within a certain range. Instead of broadcasting the entire information into the network, `Ahoy` utilizes attenuated Bloom filters to represent the availability of the context information types to neighboring nodes. It provides the availability and direction to the available information with a certain accuracy that decreases with distance to the source due to increased probability of false positives. Those space-saving filters are only broadcasted when there is a change in the network structure or in the existence of information. Meanwhile, queries are only forwarded to the nodes which most probably contain the required information based on positives in various layers of ABFs received from neighbors, instead of to all nodes. It can thus avoid flooding of information during exchange and maintenance, as may be the case in the proactive protocol, whereas it also avoids flooding of queries, as can be the case in the reactive protocol.

Figure 3.8 illustrates the relation between the amount of traffic generated and the accuracy of the advertised information in the proactive protocol and `Ahoy`. The reactive protocol is not presented here, since there is no advertisement. The height of the blocks represents the amount of traffic, and the darkness of the color represents the accuracy. The amount of traffic generated by the proactive protocol increases with the number of hops. Therefore, we obtain a shape in the form of a fan, as shown in Figure 3.8. However, the accuracy of the advertised information stays the same, as the color of the shape does not change while the number of hops increases. In `Ahoy`, the amount of traffic stays constant even when the number of hops increases, visualized as a rectangular shape in Figure 3.8(b). However, the accuracy decreases, shown by the fading color in Figure 3.8(b) from left to right, when the number of hops increases.



(a) The proactive protocol (b) Ahoy

Figure 3.8: Visualization of the amount of traffic and accuracy of the advertised information by the proactive protocol and `Ahoy`. The height of the blocks represents the amount of traffic. The smaller the height, the less traffic there is. The length of the blocks represents the number of hops, starting from 0. The darkness of the color represents the accuracy. The darker the color, the more accurate the advertised information.

We describe in detail how `Ahoy` works in the remainder of this chapter, and evaluate the performance of `Ahoy` in Chapter 4, 5, and 6.

3.3 Context Exchange

During context exchange, nodes learn about the existence and the location of information in reach. ABFs are used to store context type information. Information is hashed into Bloom codes and filled in the ABFs. ABFs represent the information layer by layer. Layer i contains the related information i hops away. ABFs are aggregated and distributed through the network. After several exchanges between neighboring nodes, each node has updated ABFs, which indicate which information is in which directions within a certain number of hops. In the following part of the section, we first introduce the fundamental operation, *context aggregation*. Then, we continue with the details of context exchange and discuss the design choices we encounter.

3.3.1 Context Aggregation

Context aggregation is a fundamental operation on attenuated Bloom filters, which is used for context exchange in `Ahoy`. It is used to combine information from a node itself and from its neighbors. The outcome of context aggregation is an attenuated Bloom filter which contains all the information that the node can reach. This final ABF is the outgoing ABF of the node, which is broadcasted to its neighbors.

Context aggregation *is the action to combine incoming attenuated Bloom filters from neighboring nodes and an ABF that contains local context information types of the node itself.*

The first layer of this new outgoing filter should be filled with the local context information from the node itself. The filters from the neighbors are shifted one layer down. The new filter represents the union of the shifted filters. Table 3.3 presents the pseudo-code for the operation. Its arguments are the local filter ($filter\_local$), received filters from the k neighbors ($filter\_in[1,..]$, $filter\_in[2,..]$, ..., $filter\_in[k,..]$), and the depth d of the filters. The first layer (layer 0) of the resulting filter ($filter\_out$) contains the first layer of the local filter ($filter\_local$). Subsequent layers of the resulting filter are constructed by applying an OR operation on bits of the shifted incoming filters and its local filter. Note that the last layer of the incoming filters is not used and will be discarded. As a result, the first layer of

$filter\_out$ (layer 0) represents the local information, the second layer contains the information from direct neighbors, and the third layer covers the information two hops away, which can be reached via direct neighbors, etc.

Table 3.3: Pseudo-code for ABFs aggregation.

```
1  aggregation (filter_local, filter_in[1,..], filter_in[2,..], ..., filter_in[k,..], d)
   {
2      % The first layer of the outgoing filter is the first layer of local filter
3      filter_out[0] = filter_local[0];
4      % Subsequent layers of the outgoing filter are constructed by shifting the incoming
5      % filters one layer down,and applying an OR operation on corresponding bits.
6      for i = 1 to (d − 1) {
7          filter_out[i] = filter_local[i]|filter_in[1, i − 1]|filter_in[2, i − 1]|...|filter_in[k, i − 1];
8      }
9      return filter_out          % return the new filter
10 }
```

Figure 3.9 exemplifies the context aggregation operation for a node with two neighbors. In this case, no local information is duplicated into the lower layers of $filter\_local$, which therefore consist of 0s. Context duplication is addressed further in detail in Section 3.3.3. In this example, each node has an ABF with 8 bits, i.e., width $w = 8$ and 3 layers, i.e., depth $d = 3$. The node uses two independent hash functions, i.e., $b = 2$ to encode its local context sources "temperature" and "humidity" into {2,8} and {2,5} respectively. The context information type is represented in $filter\_local$, as shown in Figure 3.9. When the node receives the incoming filters $filter\_in[1, ..]$ and $filter\_in[2, ..]$ from its neighbors, it shifts the received filters one layer down and discards the last layer of those filters. Thus, $filter\_in[1, ..]'$ and $filter\_in[2, ..]'$ are obtained. We perform a logical OR operation on each set of corresponding bits of $filter\_local$, $filter\_in[1, ..]'$, and $filter\_in[2, ..]'$ to obtain $filter\_out$. $filter\_out$ is the ABF that the node broadcasts to its neighbors.

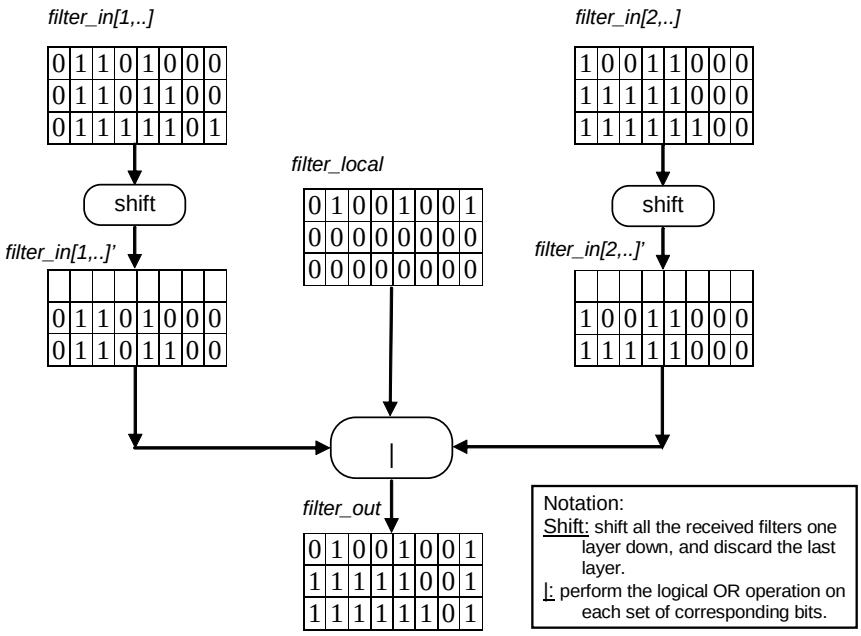


Figure 3.9: An example of ABFs aggregation without duplication.

3.3.2 Context Exchange

Every node stores two kinds of ABFs: incoming ABFs from each direct neighbor and an aggregated outgoing one. Whenever a new network needs to be established, which is the case when the `Ahoy` system is initiated, every node generates an ABF with its local information and broadcasts it. Each broadcast is identified with an generation identity, GID. In this first ABF, the first layer contains the local context information types, and the other layers are either blank or contain the same local information as the first layer, depending on whether we perform context duplication or not (please refer to Section 3.3.3 for more details). A timer is used for sending out keep-alive messages, which is explained in more detail in Section 3.5.

All nodes follow the 3 steps that are introduced below when they receive an ABF.

Step 1: When nodes receive ABFs from their neighbors, they store the ABFs. Every node stores the latest incoming ABFs for each neighbor. If one of the newly

received ABFs is different from the former one, the node updates the out-dated filter with the latest version.

Step 2: Nodes perform "aggregation" operation over all the incoming filters, as elaborated in Section 3.3.1, when they perform an update. The outcomes of "aggregation" are represented by the outgoing ABF.

Step 3: Nodes broadcast the updated outgoing ABFs, if the outgoing ABF has changed as a result of Step 2.

A node automatically stops this when no more updated ABFs are received from neighbors. By then, it has a clear overview of the present context information types around.

This procedure is also applied to brand new nodes that start to participate in an existing network. Table 3.4 depicts the related pseudo code of the functions for initialization, and receiving and updating ABFs.

Table 3.4: Pseudo-code for initialization, and receiving and updating ABFs.

```
1   initial {                                          % initialization of a new node
2       local_ABF = init_ABF(all_local_information);   % generate the local ABF
3       own_ABF = aggregate(all_ABFs);                 % generate the outgoing ABF by aggregating all ABFs
4       GID = 1;                                       % set generation ID
5       send_ABF(own_ABF, GID, all);                   % broadcast the outgoing ABF
6       timer = keep_alive_period;                     % start counting the timer for keep-alive messages
7   }
8
9   receive_ABF (receive_from, ABF_packet) {
10      if is_not_known_neighbor(receive_from) {       % not from an existing neighbor
11          add_neighbor(receive_from);                % add new neighbor
12      }
13      update_ABF(receive_from, ABF_packet);          % update all ABFs
14      if own_ABF <> own_ABF_last {                   % if there is change in the out-going ABF
15          GID++;                                     % increase GID
16          send_ABF(own_ABF, GID, all);               % broadcast the updated ABF
17          timer = keep_alive_period;                 % start counting the timer for keep-alive messages
18      }
19  }
20
21  update_ABF(receive_from, ABF_packet) {             % update the received ABF for a specific neighbor
22      if ABF_packet <> ABF_last(receive_from)        % if it is different from the last one
23          ABF_last(receive_from) = ABF_packet;       % replace the last one
24      }
25      own_ABF = aggregate(all_ABFs);                 % generate the outgoing ABF by aggregating all ABFs
26  }
```

Example scenario

Figure 3.10 depicts a step by step example when a node joins an existing network. Node A travels to a new network environment consisting of Node B, C, D, E, and F, as shown in Figure 3.10(a). Node A sends out the filter BF_A containing its own local information, as is shown in Figure 3.10(b). Node B and C are the direct neighbors of A and they receive BF_A and store it, which is shown in Figure 3.10(c). B and C aggregate A's filter into their own outgoing filters BF_B and BF_C, respectively, and broadcast them, as is shown in Figure 3.10(d). A receives the updates from B and C and aggregates those two filters into the existing outgoing filter BF_A. A broadcasts the updated BF_A. Meanwhile, Node D and G, as the direct neighbors of B and C, receive the updated BF_B and BF_C from them. They compare with their existing filters. If there is any update found in the new filter, the old one(s) will be renewed and broadcasted, which is shown in Figure 3.10(e). The updates continue till no new information needs to be added into the filters.

3.3.3 Design Choices

There is a basic assumption, which is necessary to complete the context exchange: nodes have the knowledge of the hash functions and the standards of the ABFs in use, i.e., the width, w, and the depth, d. The number of hash functions, b, the width, w, and the depth, d, are highly dependent on the context information density, and the frequency of updates and queries. We address this issue in detail in Chapter 4.

Meanwhile, we encounter a design choice whether or not to duplicate local context information types in every layer of the outgoing ABF when a node first broadcasts its information. We address this issue below.

Context Duplication

Context exchange is a process in which reachable context information types are filled in every layer of the ABFs. The information i hops away is stored in layer i. If a node is not isolated and has some neighbors, the node can in theory reach its own local information via a neighbor in multiple hops. For example, via one neighboring node, a node can reach its own information in any even number of hops. As a result,

after a number of iterations, its outgoing ABF will contain this information in all even layers. Similarly, via multiple neighbors, this information can be presented in other layers as well. As a result, a new node needs to update its outgoing filter a few times to fill its local information layer by layer. Each update might also trigger neighboring nodes to update. Nodes keep on exchanging ABFs to fill their outgoing filters with local information from the new node. We name this an "advertisement loop" in this thesis.

To avoid this to happen, an advertisement timer can be set for each node. When a new context information type is announced by a node, neighboring nodes update their ABFs accordingly. By keeping on exchanging all the updates with each other during a short period, the new information types are added into all corresponding layers of their ABFs. A node might receive updates from several neighbors in a very short time. The number of updates can be reduced, if nodes wait for a while and aggregate multiple changes into one update. This can be done by setting an advertisement timer. A node can only send one advertisement within the period of the timer. In this way, nodes aggregate changes that occurred within one period of the advertisement timer into one ABF and broadcast it. Ideally, when the timer is larger, nodes can aggregate more changes into one outgoing ABF. However, even if the timer is set to infinite, a node can not always complete the update at once. Some of the updates are dependent on each other. For example, a node needs to add new information into layer i of its ABF. When the neighboring nodes receive this update, they can add the new information into layer $(i + 1)$ of their ABFs. Only when the node receives the updates of its neighbors, it can add the same new information type into layer $(i + 2)$ of its ABF. Although the advertisement timer is a simple solution, it is not very effective to solve the problem completely.

Therefore, we propose another solution, which is to duplicate the local information into all lower layers of the ABF. For instance, there are two nodes A and B, which are direct neighbors. They form a new network and need to add each other as a neighbor. By duplicating their information into the lower layers of their ABFs, the nodes can avoid that an advertisement loop occurs. Figure 3.11(a) and Figure 3.11(b) show the sequence diagram of the packet exchange between A and B without and with duplicating local information, respectively, when d equals 4. Node A contains information 'A' and node B contains information 'B'. In this example,

A and B reach stable states where no more context exchange is necessary, after 5 packets exchanges without duplication, and after only 3 packets exchanges with duplication. Using duplication can save almost half of the packets that are exchanged for adding information in the lower layers. In general, the more layers are defined in the ABFs, the more packets can be saved by duplication.

Note that in this example, the final filters of A and B are different for the two cases. More information is included in the filters when the local information is duplicated, which contributes to a higher false positive probability. This is however only the case for a network with two nodes. In reality, when more nodes are involved, a node can reach itself via multiple paths. Therefore, the variety in the number of hops is increased. For a high node density, the final states of the two cases will be more or less similar to each other. For example, suppose that three nodes A, B, and C, are connected to each other, as is shown in Figure 3.12(a), A can reach itself through the paths {A, B, A}, {A, C, A}, {A, B, C, A}, and so on. Nodes A, B and C contain information 'A', 'B', and 'C', respectively. The final outgoing ABF of node A, with and without duplication, are shown in Figure 3.12(b). Figure 3.12(b) shows that with duplication there is only extra information in the second layer of the filter, which is the information 'A'. As a result, the influence on the false positive probability is negligibly small.

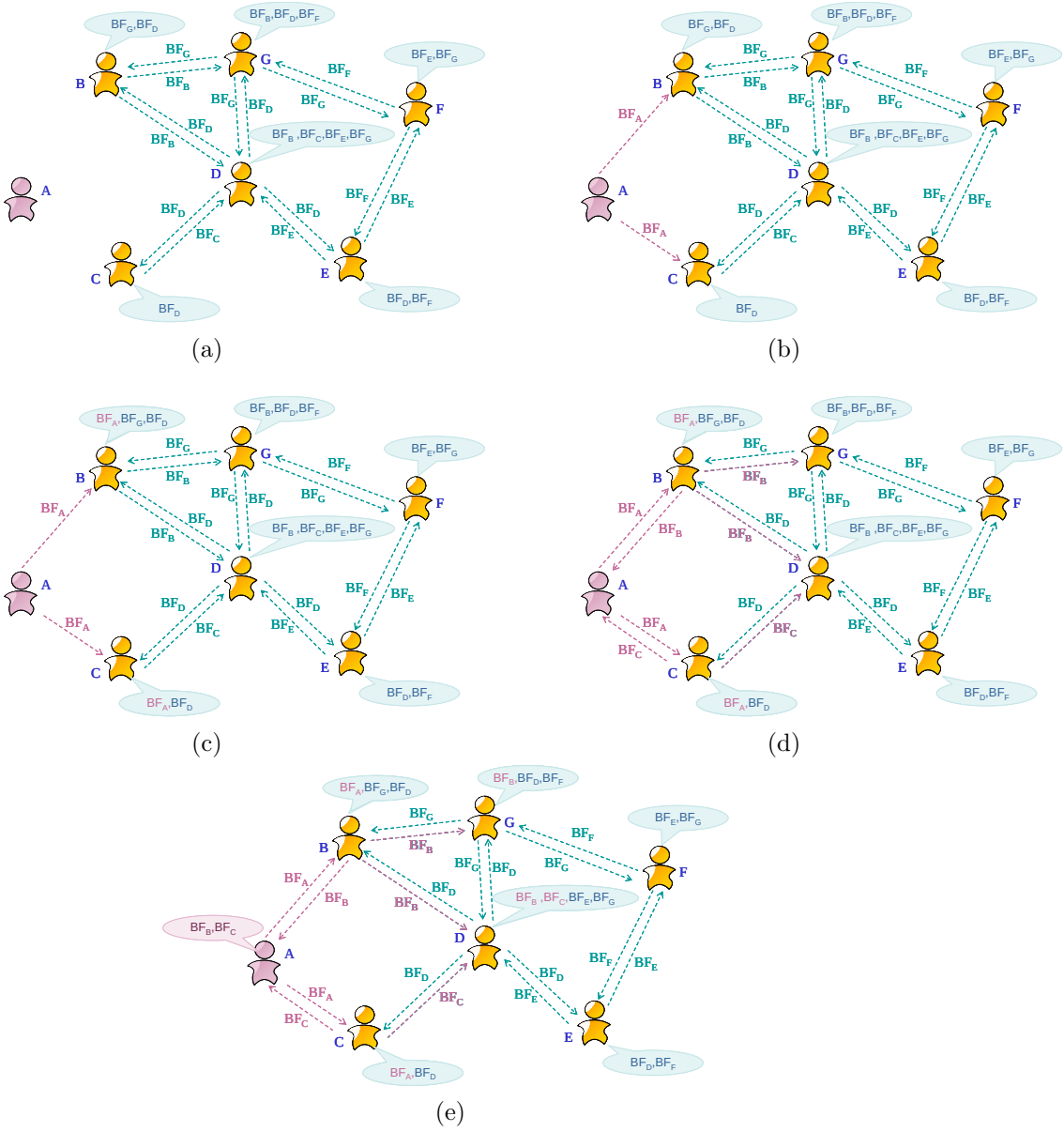
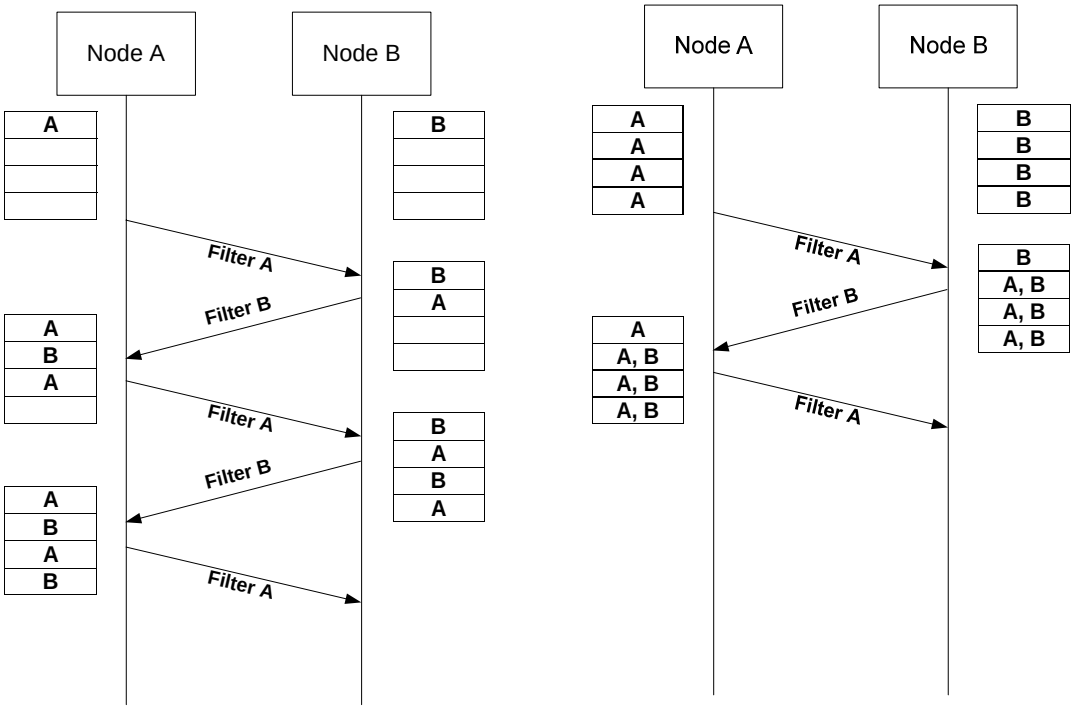
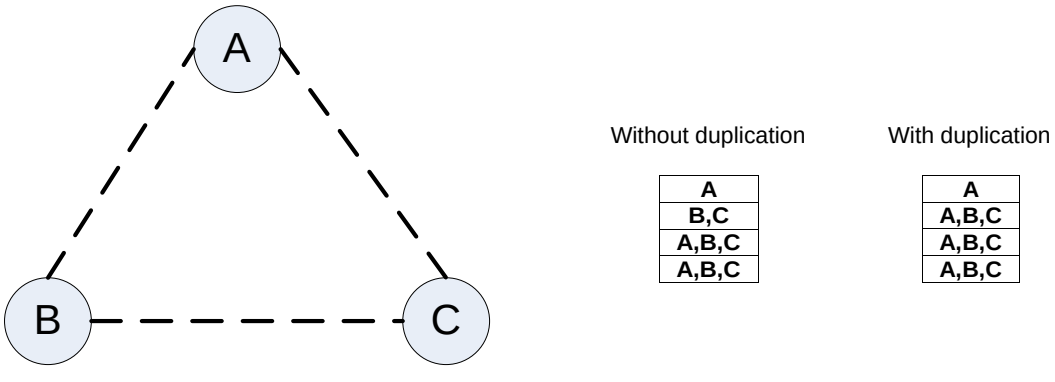


Figure 3.10: Context Exchange: a new node joins the network. The purple color indicates the updates in the figures.



(a) Without duplication (b) With duplication

Figure 3.11: Sequence diagram of packet exchange without(a) and with duplications(b).



(a) Network topology (b) The outgoing ABFs of node A

Figure 3.12: An example of three nodes network.

3.4 Context Query

Whenever a node looks for specific context information, a query is generated. In `Ahoy`, queries are compared with the incoming filters from all neighbors and only forwarded to the ones that can reach the requested information within the required number of hops. The method is named "*directional probabilistic querying*". In this section, we address the detailed query process and discuss our design choices.

3.4.1 Context Query

There are three elementary parameters used in context querying:

- QID**:** Each query is identified by an unique query identification number, QID. It is used to reduce the redundant traffic generated due to multiple alternative paths. Every node keeps a list of queries (QID) that it receives. If a node receives queries with the same QID from the same originator address more than once, it processes the first one it receives and drops the others.

- *hop_count***:** A hop counter, *hop_count*, counts the remaining number of hops a query can travel further. It is used to restrict the query range. Queries can only be sent a limited number of hops away from the node that is querying, which typically equals the depth of the ABFs, d.

- *time_out***:** *time_out* is a timer to count down the expiration of the querying period. When a node generates a query, the node also sets a *time_out* value and it starts counting down. If no reply is received when the *time_out* value reaches 0, the query is expired and no required information has been found.

A node follows the following steps to **send** a query:

Step 1: A querying node first searches locally whether it has the required information.

Step 2: If the required information is not available locally, the information is hashed into a Bloom filter (BF) and compared to the stored neighboring ABFs.

Step 3: If there is no match, the query is discarded. If a match is found in the ABF of a certain neighbor, a query message is sent to that specific neighbor with *hop_count* set to d and a unique QID. The querying node starts a timer (*time_out*).

When a node **receives** a query, it follows the following steps to process the query:

Step 1: It checks first whether the query has been received before, based on QID and the query originator address. If yes, the query message is discarded; if not, the receiving node performs following steps.

Step 2: It checks the query against the locally available context sources. If there is any match on the first layer, a reply message is sent back to the querying node. If there is no match, it takes Step 3.

Step 3: The receiving node decreases the *hop_count* by 1 and checks the first *hop_count* layers of the incoming ABFs from the direct neighbors of this node. Whenever there is a match and the *hop_count* is larger than 0, the query is propagated to that node. If no match is found, e.g., because of a false positive match in the previous hop, or the *hop_count* equals 0, the query is discarded.

If no response is received by the originating node within a time-out period (*time_out*), it understands that the required information is not available in the range of d hops.

3.4.2 Design Choices

There are three design choices that we need to make with respect to querying.

- Query format: original information format vs BF;
- Query method: parallel vs sequential;
- Record query route.

In the following part of the section, we discuss these in detail.

Query format

A query can be sent to the neighbors in two formats: an original information format or a basic Bloom filter (BF). The original information format can be text strings, XML, etc, dependent on the details of the protocol design. In terms of packet size, a packet containing the original information format, such as a text string or XML, is generally larger than one containing a BF. For the same amount of information,

using the original information format needs a much larger packet than a BF. Sending a BF query directly from the query initiator has also the advantage that hashing of the same information only needs to be done once rather than multiple times in the intermediate nodes along the query path.

However, a BF query might lead to false reply messages if the BF matches other context type information than the requested one. This occurs when two or more context information types are hashed into the same Bloom codes, which in the original format, is not possible. Choosing proper hash functions can minimize the chance that the mismatch happens in BF query format (this is elaborated in more detail in Section 6.6.1).

Another important disadvantage of BF queries is related to security. It offers the possibility of generating malicious traffic by propagating these queries through the network. In Section 6.4.1, we argue that querying with the original format can well solve this problem.

Further, both query formats have different vulnerability against various malicious attacks, which we will address in detail in Chapter 6. The final choice of query format can be made based on the detailed network scenarios and the preference of the network designers.

Query Method

Context querying can be done by exploring possible context sources in a parallel or sequential way. In a parallel query method, a query will be forwarded to *all* the neighbors that show matches in their corresponding ABFs. Therefore, all requested information available in d hops can be found simultaneously. It offers choices to end users. In contrast, queries are only forwarded to *one* corresponding neighbor *at a time* in the sequential query method. If the queried neighbor does not have the requested information, the query needs to be forwarded to another neighbor. Consequently, only one reply can be received at once.

Table 3.5 presents pseudo-code for parallel querying. The function $receive\_query$ is called when a query packet is received from a local application or from one of the neighboring nodes. Its arguments are the sender of this received packet ($received\_from$), and the query packet itself ($query\_packet$).

Table 3.5: Pseudo-code for parallel queries.

```
1   receive_query (receive_from, query_packet) {
2       if is_not_in_id_cache(query_packet.id, query_packet.originator) {      % if query has not been received before
3           add_to_id_cache(query_packet.id, query_packet.originator);
4           if have_local_match(query_packet) {                                % check local match
5               reply_packet = make_reply(query_packet);                       % send reply
6               send_reply(received_from, reply_packet);
7           }
8           query_packet.hop_count − −;                                        % decrease hop counter
9           if query_packet.hop_count > 0 {                                    % not yet reach the maximum query range
10              for k = 1 to number_of_neighbors {                             % check all the neighbors
11                  if k <> receved_from {                                     % but not one the query received from
12                      for i = 0 to query_packet.hop_count − 1 {              %check related layers
13                          if (query_packet.bf&filter_in(k, i)) == query_packet.bf {   % match found
14                              add_to_route_cache(query_packet, received_from);        % record the path
15                              send_query(k, query_packet);                   %forward the query to neighbor k
16                              break;                                         % stop processing other layers for neighbor k
17                          }
18                      }
19                  }
20              }
21          } else {                                                           % reach the maximum query range
22              discard(query_packet);
23          }
24      } else {
25          discard(query_packet);
26      }
27  }
```

In case of sequential querying, the originating node checks the stored attenuated Bloom filters on a layer-by-layer basis. It checks one layer of every incoming ABF. If no match is found, it goes to the next layer. If a match is found in a certain filter at layer i, the search is stopped and a query message is sent to the corresponding neighbor. There are two choices of setting *hop_count*. It can be set to d, as the maximum query range. It can also be set to $i + 1$, because we expect the context source node locates i hops away from the originator, which is a more traffic-efficient way. Any node that receives a query performs in the same way. The *hop_count* first decreases by 1. It checks locally whether there is a match. If a match is found, a reply message to commit the existence of the requested information is sent back to the query originator. If not, it checks the stored ABFs on a layer-by-layer basis. The query is forwarded to the neighbor in which an ABF match is found in one of the layers 1 till *hop_count*. The query is forwarded till *hop_count* equals 0. In this case, a reply message to indicate that no information is found is sent back to the previous node. When the previous node receives such a reply, it checks other neighbors, and possibly additional queries will be sent. If not, it forwards the reply to its previous node. All nodes who receive such a reply perform the same, until the reply reaches the query originator. Then, the query originator checks the next possible neighbor. If every possible neighbor has been checked and no successful reply has been received, it knows the requested information does not exist. Table 3.6 presents the pseudo-code for sequential querying.

Table 3.6: Pseudo-code for sequential queries.

```
1   receive_query (received_from, query_packet) {
2       if is_not_in_id_cache(query_packet.id, query_packet.originator) {      % if query has not been received before
3           add_to_id_cache(query_packet.id, query_packet.originator);
4           if have_local_match(query_packet) {                                % check local match
5               reply_packet = make_reply(query_packet, TRUE);                 % send reply
6               send_reply(received_from, reply_packet);
7           } else {
8               query_packet.hop_count − −;                                    % decrease hop counter
9               if query_packet.hop_count > 0 {                                % not yet reach the maximum query range
10                  search_next_match(query_packet, 1, 0, received_from);      % look for next match
11              } else {                                                       % reach the maximum query range
12                  reply_packet = make_reply(query_packet, FALSE);            % reply NO MATCH
13                  send_reply(received_from, reply_packet);
14              }
15          }
16      } else {
17          discard(query_packet);
18      }
19  }
20
21  search_next_match (query_packet, k, i, received_from) {                    % look for the next match
22      % start from layer i of the incoming filters of neighbor k
23      while (i < query_packet.hop_count − 1){                                % check all the corresponding layers
24          while (k ≤ number_of_neighbors)&(k <> received_from) {             % check all neighbors
25              if (query_packet.bf&filter_in(k, i)) == query_packet.bf {      % match found
```

```
26                  add_to_route_cache(query_packet, received_from);          % record the path
27                  add_to_query_cache(query_packet, k, i,
28                                  query_packet.hop_count, recived_from);    % record the query
29                  send_query(k, query_packet);                              % forward the query to neighbor k
30                  STOP;                                                     % stop the entire function
31              }
32              k++;
33          }
34          i++;
35      }
36      reply_packet = make_reply(query_packet, FALSE);                       % reply NO MATCH
37      send_reply(received_from, reply_packet);
38  }
39
40  receive_reply (reply_packet) {                                            % receive a reply
41      query_packet = reply_packet.query_packet;
42      if reply_packet.reply == TRUE {                                       % find the requested information type
43          remove_query_cache(query_packet);                                 % remove from cache
44          if is_originator(query_packet) {                                  % is the query originator
45              find_context(query_packet);
46          } else {                                                          % not the query originator
47              received_from = retrieve_data_route_cache(query_packet);
48              send_reply(received_from, reply_packet);                      % Reply to the neighbor that sent the query
49          }
50      } else {                                                              % the requested information type not found
51          (k, i, hop_count, received_from) =                                % retrieve stored data
52                      retrieve_data_query_cache(query_packet);              % retrieve stored data
53          query_packet.hop_count = hop_count;
```

```
54          remove_query_cache(query_packet);                          % remove from cache
55          remove_route_cache(query_packet);                          % remove from cache
56          search_next_match(query_packet, k + 1, i, received_from);  % look for next match
57      }
58  }
```

In general, a longer query time is consumed in the sequential method, compared to the parallel method. The same amount of time is used, only if the first querying path is the one through which the requested information is reached. If it is not, more query paths are visited, and it takes a longer time to find the information. However, with the sequential query method, fewer nodes are queried compared to the parallel method. In the best case, only nodes along one path are queried. In the worst case, all possible nodes are queried, which is the same as with the parallel method. The sequential method therefore saves some traffic in this respect. But it also generates more replies. No matter whether the requested information type is found or not, at the end of each path, a reply is generated. However, the sequential method also has an advantage. It can specify different number of query hops for each neighbor, which in the parallel query is always set to a same number in one broadcast message to all neighbors. In case the parallel query method would also like to set different numbers for different neighbors, it needs to unicast different queries to each neighbor. In this respect, the sequential query method is slightly more efficient, if we know in advance within how many hops we expect to find the queried information.

The choice between parallel and sequential queries is a trade off between the number of replies, the used bandwidth, and the response time. With the sequential method, at most one node is found that possesses the requested information; less traffic is generated; but in general, more querying time is consumed. Using parallel querying, all the nodes with the requested information are queried; more traffic for querying is generated; but in general, less querying time is used. The choice highly depends on the specific network scenario, such as the network density and user or vendor requirements. For instance, in a very high density network, parallel queries might generate heavy traffic, since multiple neighbors are queried simultaneously. Therefore, in high density networks, we suggest to use the sequential query method to avoid flooding the network with query messages. In low density networks, we suggest to use parallel query method to save querying time.

Route Recording

When a node has the requested information, it needs to send a reply to the node that has sent the query. This can only be done if the (shortest) path to the querying

node is known.

There are three alternatives for recording query routes for providing the (shortest) path. The first option is to maintain (soft) state routing information in the nodes. Nodes record where a query is received from and sent to. The reply of such a query is routed back to the query originator based on those information. This is shown in the pseudo codes for parallel and sequential queries. This can put an extra burden on those nodes that store the routing information.

Alternatively, the reply path can be stored in the query messages themselves. This results in a larger query packet size and increased transmission costs. In the first two alternatives, the reply message is sent along the same path as the query (but in opposite direction). They both encounter the problem that the reply message gets lost, whenever a node along the path moves or disappears. However, they have the advantage that no extra routing functionality has to be present in the platform.

The third alternative is that the system can rely on an external routing protocol. This has the benefit of no consumption of extra storage room or bandwidth. It does not rely on the same paths used in Ahoy. Even if the path to the querying node has changed, the reply message can still reach the querying node as long as there is at least one path to that node. However, this alternative requires extra functionalities of routing protocols. Since a routing protocol is mostly needed to set up a connection between the querying and replying node to deliver the requested information, it could be a good option after all.

The final choice is again based on the design preference and different network scenarios.

3.5 Context Update and Maintenance

`Ahoy` needs a mechanism to support nodes maintaining and updating the information in the network. Nodes should be aware of the existence of their neighbors, and be able to detect the appearance and disappearance of neighbors and their context information. `Ahoy` utilizes a **keep-alive mechanism** to keep nodes alert for changes in their environment. In the mechanism, each outgoing ABF of a node has a unique generation identification, called GID. The GID is incremented by 1 every time a new ABF is sent by the node. A node stores the latest ABF and its GID for each

neighbor. A **keep-alive message** is a short message which contains the latest GID. The keep-alive period is the fixed time interval between two consecutive keep-alive messages. It is assumed that it is determined in advance, and that the same value should be used over the entire network. Each node in the network therefore should know the keep-alive period.

The keep-alive mechanism operates always 5 types of actions:

1. *A node sends out keep-alive messages periodically, when there is no change in its incoming or outgoing ABFs. A timer is used to determine the time stamp on when the keep alive messages are sent. Whenever an ABF or a keep alive message is sent out, the timer is set back to the keep alive period. When the timer expires, a keep alive message is sent out.*

2. *A node aggregates all ABFs from its neighbors, whenever there is a change in its local context information or in the incoming ABFs of its neighbors, including a new ABF from a newly discovered neighbor, or no ABF anymore from a disappearing neighbor. The newly generated outgoing ABF is broadcasted if it is different from the last outgoing ABF.*

3. *If a node receives a keep-alive message from an existing neighbor with a GID newer than the one stored for that neighbor or a keep-alive message from an unknown node, it sends out an update request to that node.*

4. *A node replies to an update request by broadcasting its latest outgoing ABF. If the update request is from an unknown node, the node adds this node as a new neighbor.*

5. *A node removes a neighbor and its related information if it does not receive any ABF or keep-alive message from this neighbor within two consecutive keep-alive periods.*

Table 3.7 represents the above mentioned 5 actions by pseudo code. Action 1 is illustrated in Function *send_keep-alive*. Action 2 is depicted in Function *receive_local_change* and *receive_ABF* from Table 3.4. Action 3 is related to Function *receive_keep_alive*. Action 4 is explained by Function *receive_update_request*. Action 5 is addressed by Function *remove_neighbor*.

Table 3.7: Related pseudo-code for context update and maintenance.

```
1   send_keep-alive {
2       timer − −;
3       if timer == 0 {
4           send_KA(GID, all);                          % broadcast KA message with the current GID
5           timer = keep_alive_period;                  % start counting the timer for keep-alive messages
6       }
7   }
8
9   receive_local_change (local_info, add/remove){      % receive local information change
10      update_local(local_info, add/remove);
11      if own_ABF <> own_ABF_last {                    % if there is change in the out-going ABF
12          GID + +;                                    % increase GID by 1
13          send_ABF(own_ABF, GID, all);                % broadcast the updated ABF
14          timer = keep_alive_period;                  % start counting the timer for keep-alive messages
15      }
16  }
17
18  update_local (local_info, add/remove){              % updates due to local information change
19      if add {                                        % add information
20          local_ABF = add(local_info);
21      }
22      elseif remove {                                 % remove information
23          local_ABF = remove(local_info);
24      }
25      own_ABF = aggregate(all_ABFs);
```

```
26  }
27
28  receive_keep-alive (receive_from, keepAlive_packet) {          % receive KA message
29      if is_known_neighbor(receive_from)
30          && is_same_GID(receive_from) {                         % from an existing neighbor with the correct GID
31          update_last_time(receive_from, keepAlive_packet);      % update the last receive time
32      }
33      else {                                                     % from an unknown neighbor
34                                                                 % or an existing neighbor with different GID
35          send_update_request(receive_from);                     % send update request
36      }
37  }
38
39  receive_update-request (receive_from, updateRequest_packet) {
40      if is_not_known_neighbor(receive_from) {                   % if it is not from an existing neighbor
41          add_neighbor(receive_from);                            % add new neighbor
42      }
43      send_ABF(own_ABF, GID, all);                               % broadcast its ABF
44      timer = keep_alive_period;                                 % start counting the timer for keep-alive messages
45  }
46
47  remove_neighbor {
48      for i = 1 to numberofneighbors {
49          if current_time − last_time_KA/ABF(i) >= 2 · keep_alive_period {   % time out
50              remove_neighbor_list(i));                          % remove neighbor i from the list of neighbors
51              own_ABF = aggregate(all_ABF);                      % aggregate the remaining ABFs
52              if own_ABF <> own_ABF_last {                       % if there is change in the out-going ABF
53                  GID++;                                         % increase GID by 1
```

```
54                  send_ABF(own_ABF, GID, all);          % broadcast the updated ABF
55                  timer = keep_alive_period;            % start counting the timer for keep-alive messages
56              }
57          }
58      }
59  }
```

There are five cases that could happen in context updates and maintenance, as addressed below:

Case 1: If there is *no change* in the context, keep-alive messages are sent out periodically based on **Action 1**. A node can identify the freshness of the stored ABFs by comparing GIDs. Once it notices that a GID is different from that of a stored ABF, it knows that one or more updates might have gotten lost. An update request is sent out based on **Action 3**. The neighbor replies with its latest ABF, based on **Action 4**. If the ABF contains different information than the stored ABF, the node replaces the stored one with the new ABF and aggregate all ABFs. A new ABF is generated and broadcasted to the neighbors based on **Action 2**.

Case 2: If there is *any change* in the possessed context information, a node sends out the updated ABFs based on **Action 2**. Whenever a node receives a different ABF compared to the one stored from the corresponding neighbor, it replaces the stored ABF with the new one and aggregates all ABFs from all neighbors and broadcasts it based on **Action 2**. The change is propagated among other nodes, till all nodes in the vicinity (e.g., d hops) of the node, are updated.

Case 3: If a node receives a keep-alive message from an *unknown neighbor*, it realizes that a new node comes into its communication range. It sends out an update request to the new neighbor based on **Action 3**. The new neighbor replies with its latest ABF based on **Action 4**. The node adds the new neighbor into its neighboring list and aggregates all ABFs including the new one. The newly generated ABF of the node is broadcast to the neighbors based on **Action 2**. Similar actions are taken by the new neighbor. The other neighbors of the node might also need to update their ABFs a couple of times to coordinate the change based on **Action 2**. This updating process will be further addressed in detail in Section 5.2.4 and Section 5.3.3.

Case 4: If a node receives an ABF from an unknown neighbor which just moves into its communication range, it adds the node as its new neighbor, aggregates the information into its out-going ABF, and broadcasts it based on **Action 2**. The new neighbor and the other existing neighbor(s) react accordingly based on **Action 2**, until there is no change in the out-going ABF of any node.

Case 5: If a node *does* ***not*** *receive either keep-alive messages or ABFs from a certain neighbor for* ***two consecutive*** *keep-alive periods*, it considers the neighbor

has left and removes it from the neighboring list based on **Action 5**. This action triggers the node to re-aggregate the ABFs from the existing neighbors, from which any information related to the disappearing node is also removed. The deletion of the disappearing node is propagated further to the other neighbors if necessary based on **Action 2**. This will be further discussed in detail in Section 5.2.3 and Section 5.3.2.

3.6 Discussion

`Ahoy` is a context discovery protocol especially designed for ad-hoc networks. The differences between `Ahoy` and the proactive and reactive protocols are summarized in Table 3.8.

Table 3.8: The differences between Ahoy, the proactive and reactive protocols.

	Context exchange	**Context query**	**Context update and maintenance**
Ahoy	Broadcast with ABFs within d hops	Directional queries to the neighbors who probably have access to the required information	Update upon changes within d hops and maintain with keep-alive messages
Proactive	Broadcast to part of or the entire network	Query only the node who has the required information	Updates in part of or the entire network
Reactive	No	Query over part of or the entire network	No

Based on the characteristics of ad-hoc networks, summarized in Section 1.2, a discovery protocol is needed that can locate the requested information without frequent packet exchange and complicated processing. `Ahoy` can fulfill these requirements. The following two features are essential in doing so:

- **To utilize ABFs to save space and computation complexity.** Compared with the text strings or XML format used by the proactive protocol,

the ABF format requires much less storage and transmission bandwidth by compressing multiple context information types into a small ABF and still clearly indicate the existence of the information with reasonable accuracy. To aggregate information nodes by using ABFs requires only a bit-wise OR operation. To check the availability of the information requires only a bit-wise AND operation and a comparison with results.

- **To provide directional probabilistic querying to save querying traffic.** Unlike flooding queries into the entire network, nodes can only forward queries to the neighbors that with high probability contain the requested information.

We can imagine the proactive protocol is perfectly suitable for a static network with frequent queries. Nodes only need to exchange their information once and can find the requested information immediately without searching. On the other hand, the reactive protocol should perform best for a highly mobile network with occasional queries. When the information in the network keeps on changing, it consumes a lot of traffic to keep all nodes updated. The best option will be to look for the requested information only when necessary. However, those two are extreme cases, either totally static or highly mobile. In the more general situation where nodes in the network move and look for certain information on a regular basis, we believe `Ahoy` has a better performance than the two conventional approaches with the two space and traffic saving features.

As we introduced in Section 3.1, false positive is a drawback of Ahoy. Choosing a larger (wider) filter decreases the chance of false positive probability. However, this results in a larger packet size of advertisements. Conversely, a small size of advertisement packet can well reduce the traffic for context exchange, but increases the number of false positive queries. It is a trade-off, which depends on a large number of factors, such as the size of the attenuated Bloom filters (width and depth), the number of hash functions, and the number of context information types, advertisement and query packet size and rate. In the next chapter, we elaborate on this and compare the network traffic generated by `Ahoy` and the two conventional approaches.

Chapter 4

Performance Modeling

In Chapter 3, we introduced the light-weighted discovery protocol, `Ahoy`, for MANETs. It utilizes space-efficient context representation and selective query distribution. However, in Section 3.6, we emphasize that due to false positives, redundant queries may contribute significantly to the costs of context discovery. In order to reduce the number of unnecessary queries, we have to reduce the rate of false positives, which can only be done by using larger attenuated Bloom filters. To achieve optimal network costs, in terms of number of bits transmitted per second, we need to strike a balance between the number of false positive and the size of the attenuated Bloom filter (ABF). The optimal size of the ABF depends on its depth, the rates at which advertisements are sent and queries are generated, and the cardinality of the represented set, i.e., the number of context information types that are to be advertised.

In this chapter, we examine the network cost generated by `Ahoy`. An analytical model is set up to obtain the size of the Bloom filter that yields the minimum network costs. In the model, the width of the filters and the number of hash functions can be set to achieve the optimal network cost, and to observe how the amount of traffic in `Ahoy` compares to that in conventional approaches.

The chapter is organized as follows. In Section 4.1 we introduce the preliminaries used in our modeling. In Section 4.2, the `Ahoy` network cost function is induced. We give a brief introduction on two types of reference protocols: proactive and reactive protocols, in Section 4.3. In Section 4.4, the experimental results are presented. Finally, we validate the analytical model with the simulation model established in [27] in Section 4.5.

The work presented in Section 4.1 has been published in [53, 54]. The work in Section 4.2, Section 4.3, and Section 4.4 has been published in [51, 52, 50]. The work in Section 4.5 has been published in [50].

4.1 Modeling Preliminaries

In this section, we introduce the modeling background. In Section 4.1.1, we describe the network structures that are used in our model. To calculate the network costs, we have to quantify the packet size, the transmission rate, and the connectivity between nodes. We focus on these issues in Section 4.1.2.

4.1.1 Network structure

We use two typical network structures to model the performance of `Ahoy`. The first network has a grid structure, and the second one has a circular structure.

Grid structure

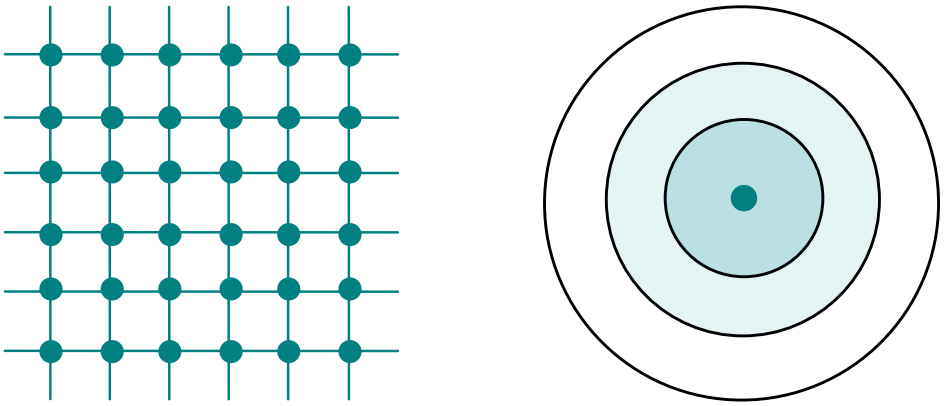
In a grid structure, each node has 4 direct neighbors within its communication range. The distance between a pair of connected nodes is equal to the communication range r. The connections between nodes can be considered as series of intersecting vertical and horizontal axes which form a two-dimensional $r \times r$ sized grid. An example is shown in Figure 4.1(a). The regularity of the grid structure and the fixed number of neighbors for each node enables us to use a rather simple model to analyze the performance of `Ahoy`. In the following part of the thesis, we use superscript g to denote the formulas for grid structured networks.

Circular structure

Networks can be represented as a graph $G = (V, E)$, where nodes are vertices (V), and links between nodes are edges (E). The structure of an ad-hoc network can often be modeled as a random geometric graph [65] (e.g., [5, 34]), which can be defined as below.

*A **random geometric graph** places vertices randomly, uniformly, and independently into a bounded region. Vertices are connected with edges if the distance between two vertices is smaller than or equal to threshold $r (r > 0)$.*
In this thesis, we use random geometric graphs as the basis for our analysis.

From the point of view of a selected node, the set of areas in which nodes are located at a distance of i hops, can be approximated by a set of concentric circles, one for each value of i. For this reason, we refer to this model as the circular structure, shown in Figure 4.1(b). In the following, the formulas related to a circular structured network will be represented with the superscript c.



(a) Grid structure (b) Circular structure

Figure 4.1: Network structures.

4.1.2 Connectivity in Ad-hoc Network Models

When we count the network traffic generated in `Ahoy`, it is essential to know how many nodes are transmitting how many packets at a certain number of hops away. The first question we need to resolve here is how nodes are connected and what is the node degree multiple hops away.

Graph theory is often used to facilitate the studies in the area of node *degree* and *connectivity*. In graph theory, the *degree* of a vertex can be defined as the number of edges incident to it. When it is applied to describe a network, the *degree* of a node can be defined as the number of direct neighbors that it has. A graph is *connected*,

if there is a path from any vertex to any other vertex in the graph. A graph is called *k-connected* if the graph remains connected when fewer than k vertices are removed from the graph.

Current research of connectivity mostly focuses on the following two major questions: (1) how to achieve a k-connected network; (2) what is the degree distribution of a node. For example, [34] studied the degree distribution of a node in an ad-hoc network through a combination of analytical modeling and simulations. They found that the degree distribution is binomial for low values of the mean degree. For a given network density and communication range, we can thus obtain the distribution of the number of direct neighbors a node has. [5] has investigated the relationship between range r, node density, and the probability that the network is k-connected, assuming random geometric graphs. The results provide the principles for choosing practical values of those parameters for simulations and design.

However, to our knowledge no research has been done on the degree distribution multiple hops away. Here, we define the i-hop node degree as being the number of different nodes a selected node can reach in exactly i hops and not in fewer hops. Let us denote the i-hop node degree as D_i.

For a grid structured network, it is straightforward to determine D_i^g. By definition, the node degree for 0 hop is:

$$D_0^g = 1. \tag{4.1}$$

For $i > 0$, Figure 4.1(a) shows that $4i$ new nodes become reachable when increasing the number of hops from $(i-1)$ to i. The ith hop node degree, D_i^g, can be derived as:

$$D_i^g = 4 \cdot i \qquad (i > 0). \tag{4.2}$$

The node degree increases linearly with distance (number of hops). The total number of nodes reachable within at most i hops, N_i^g, including the selected node itself, can be obtained by summation of the node degrees per number of hops:

$$N_i^g = \sum_{j=0}^{i} D_i^g = 1 + 2i(i+1). \tag{4.3}$$

For circular-structured network, D_i^c can be derived from D_{i-1}^c, i.e., it is conditional to the number of nodes reachable in $(i-1)$ hops. Although theoretically we

can derive D_i^c in this way, the expression is going to be computationally difficult, because the node degrees depend strongly on the specific positions of the nodes in the network. Instead, we try to estimate the expected node degree, $E[D_i^c]$. This requires the basic assumption that nodes are connected in the network. We first study under which circumstances and requirements nodes are connected and set our modeling assumptions accordingly. Then we study the multi-hop communication range. Based on the area of i-hop communication range and the node density, we obtain the expected multi-hop node degree.

Modeling Assumptions of Circular-structured Networks

First, we study the requirements for a connected network and determine the related assumptions. The network we want to model analytically is a random geometric graph, as introduced in Section 4.1.1, where N nodes are randomly and uniformly distributed in a certain area S. Although in reality the communication between two nodes is subject to various kinds of time- and place-dependent propagation effects, to simplify the analytical modeling, the communication range of nodes is often assumed as a fixed value so that a network can be modeled as a random geometric graph [5, 34]. In this thesis, we also assume for each node a fixed communication range r. The graph is connected when there is at least one path between any given pair of nodes. In contrast, when some nodes are isolated from the rest of the network, the network is called disconnected. A disconnected network contains several sub networks that cannot reach each other. According to the study from [5], the probability that the network is connected, p_c, is related to the node communication range and node density, as:

$$r \geq \sqrt{\frac{-\ln\left(1-p_c^{\frac{1}{N}}\right)}{n\pi}}, \tag{4.4}$$

where $n = \frac{N}{S}$ is the average node density, and r is the communication range. (4.4) shows that the network is connected with probability p_c, when the communication range of nodes r equals or is larger than the right side of the inequality sign, given the specified network density.

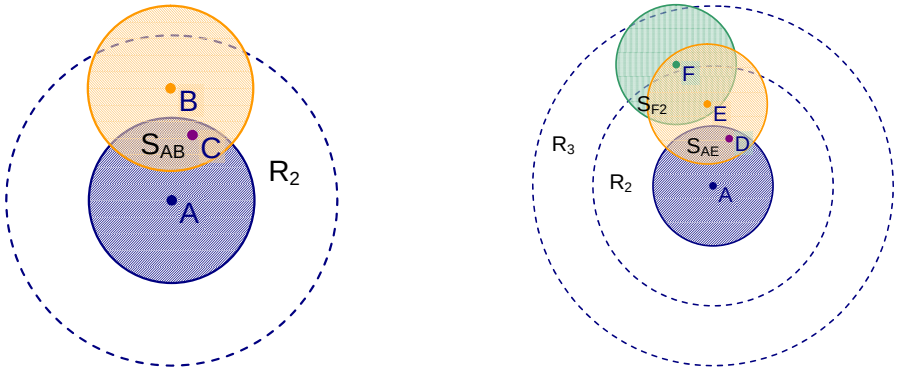
In our analytical model, we assume that the network is connected. If in reality,

some isolated nodes exist, this may have a significant influence on the network costs analysis. Nodes that are not connected do not generate extra traffic throughout the network. As a consequence, the number of exchanged packets is overestimated if the network is not connected. However, we assume **a high density network for which there is a high probability that the network is $k-$connected** ($k > 1$). For example, if the study area is 100 m^2 and the communication range r equals 300 meters, we need a node density of at least $7.45 \times 10^{-4}(\text{node}/m^2)$ to generate 1-connected graphs with 95% (confidence) probability. This density is realistic for quite some scenario's. If devices in MANETs are connected with Bluetooth or IEEE 802.11n, which have an indoor communication range of at least around 70 meters [7, 42], a network established in a normal office environment can be considered as *high* density. A network established in a conference meeting can be considered as *very high* density. A network established in a office environment after working time (when only one or two devices are switched on at one floor) should however be considered *low* density.

i-hop Communication Range in Circular Structured Networks

Then we study the area where the ith hop nodes are located. Suppose we have node A with communication range r, as shown in Figure 4.2(a). We can define concentric circles with radius ir ($i = 1, 2, 3, \ldots$) that have A as center. We define R_i to represent the annulus of the outer circle with radius ir and the inner circle radius $(i-1)r$. Node A can reach all the nodes located within a radius r. Node B is located within the annulus R_2. It is within $2r$ of A, but outside the direct reach of A. It therefore will only reach A, if and only if there is a node C located within the intersection area S_{AB} of the communication ranges of A and B. This is also illustrated in Figure 4.2(a).

The distance between A and B, d_{AB}, lies between r and $2r$. When d_{AB} equals $2r$, circle A and circle B are tangent, as is shown in Figure 4.3(a). In that case, the area of the intersection between circle A and circle B, S_{AB}, is 0. When d_{AB} is smaller than $2r$, circle A intersects with circle B. The intersection area, S_{AB} increases, while d_{AB} decreases. When d_{AB} approaches r, S_{AB} is maximum, and according to the trigonometry shown in Figure 4.3(b), the intersection area is then equal to $(\frac{2}{3}\pi r^2 - \frac{\sqrt{3}}{2}r^2)$. Hence, the intersection area of circles A and B, S_{AB}, lies



(a) (b)

Figure 4.2: (a) A and B are connected through C; (b) A and F are connected through E and D.

between 0 and $(\frac{2}{3}\pi r^2 - \frac{\sqrt{3}}{2}r^2)$.

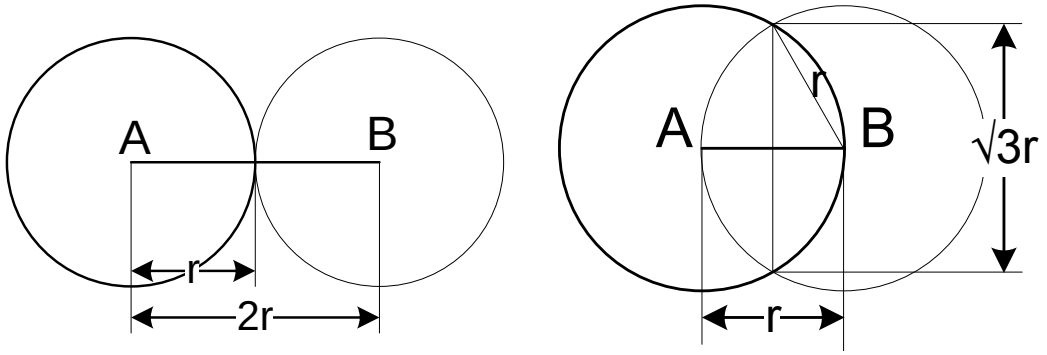
The probability that an arbitrarily chosen node lies in the intersection area, S_{AB}, is simply $p = \frac{S_{AB}}{S}$. Since we have assumed that nodes are randomly distributed in the network with average node density $n = \frac{N}{S}$, the number of nodes located in the intersection area S_{AB} is Bernoulli distributed with expectation equal to $N \cdot p$. For a large network area with high density, N is generally larger than 20, while p is smaller than 0.05. According to [33], in such a high density network, the number of nodes located in the intersection S_{AB} can be well estimated by a Poisson distribution with $\lambda_{AB} = N \cdot p = N \cdot \frac{S_{AB}}{S} = S_{AB} \cdot n$. The probability that there is at least one node located in area S_{AB}, $P(N_{AB} > 0)$, corresponds to the probability of having a path between A and B. This probability equals 1 minus the probability that no node is located in the area S_{AB}:

$$\begin{aligned} P(B \text{ is a 2-hop neighbor of } A | B \text{ is in } R_2) &= P(N_{AB} > 0) \\ &= 1 - P(N_{AB} = 0). \end{aligned} \tag{4.5}$$

Since the number of nodes is Poisson distributed, (4.5) can be rewritten as:

$$P(N_{AB} > 0) \approx 1 - e^{-\lambda_{AB}} = 1 - e^{-S_{AB} \cdot n} \qquad (0 < S_{AB} < \frac{2}{3}\pi r^2 - \frac{\sqrt{3}}{2}r^2). \tag{4.6}$$

We can observe that given an intersection area S_{AB}, the probability, $P(N_{AB} > 0)$, is a concave function of n, with $P(N_{AB} > 0) = 0$ for $n = 0$, and $P(N_{AB} > 0)$ goes



(a) (b)

Figure 4.3: (a) A and B are tangent; (b) A intersects B with $d_{AB} = r$.

to 1 when n goes to infinite. This implies that with almost 100% probability there is a path between node A and B if the node density is very high.

$$\lim_{n\to\infty} P(B \text{ is a 2-hop neighbor of } A | B \text{ is in } R_2) = 1. \tag{4.7}$$

Let us now have a look at node F in Figure 4.2(b), which is located within annulus R_3. Node F can reach node A, if and only if there is at least one node E located within the communication range of node F, which has a connection to node A. Therefore, the probability that node F is a 3-hop neighbor of A can be expressed as:

$$\begin{aligned} &P(F \text{ is a 3-hop neighbor of } A | F \text{ is in } R_3) \\ &= P(\exists E : d(E,F) \leq r \wedge E \text{ is a 2-hop neighbor of } A | F \text{ is in } R_3). \end{aligned} \tag{4.8}$$

Figure 4.2(b) shows that if node E is located outside the annulus R_2 of A, the probability is 0 that E is a 2-hop neighbor of A and a direct neighbor of F. To be a 2-hop neighbor of A, and a direct neighbor of F, E should be located within the annulus R_2 of A, and within R_1 (r) of F. Since the intersection area between R_2 of A and R of F is positive and finite, from (4.7), we can obtain the probability that F is a 3-hop neighbor of A when the network density goes to infinite:

$$\begin{aligned} &\lim_{n\to\infty} P(F \text{ is a 3-hop neighbor of } A | F \text{ is in } R_3) \\ &\Rightarrow \lim_{n\to\infty} P(\exists E : d(E,F) \leq r \wedge E \text{ is in } R_2 | F \text{ is in } R_3). \end{aligned} \tag{4.9}$$

Since n goes to infinite, we have:

$$\lim_{n\to\infty} P(\exists E : d(E,F) \leq r \wedge E \text{ is in } R_2 | F \text{ is in } R_3) = 1. \tag{4.10}$$

In a similar way, we can deduce the probability that any node X located in R_i is an i-hop neighbor of A as:

$$\begin{aligned} &\lim_{n\to\infty} P(X \text{ is an } i\text{-hop neighbor of } A | X \text{ is in } R_i) \\ =& \lim_{n\to\infty} P(\exists Y : d(Y,X) \leq r \wedge Y \text{ is } (i-1)\text{-hop neighbor of } A | X \text{ is in } R_i) \\ =& \lim_{n\to\infty} P(\exists Y : d(Y,X) \leq r \wedge Y \text{ is in } R_{i-1} | X \text{ is in } R_i) \\ =& 1. \end{aligned} \tag{4.11}$$

(4.11) demonstrates that, given node X is located in R_i, in a high-density network, X is an i-hop neighbor of A with almost 100% probability. The network is thus connected if the network density goes to infinite. Moreover, each pair of nodes is connected through a minimum number of intermediate nodes. Since node X is situated in R_i, the distance between node A and X is between $(i-1)r$ and ir. Given a communication range of r, we therefore need at least $(i-1)$ nodes to connect node A with X. The length of the (absolute) shortest path between node A and X is thus i hops.

When the network density is high enough so that the network is a k-connected ($(k > 1)$) graph with high probability, there is high chance that there exist multiple i-hop paths between the central node and most of the nodes within annulus R_i. The higher the network density is, the higher this probability is. Therefore, adding or removing a node in the network will not influence the length of the shortest path between most pairs of nodes. Suppose that there is only one shortest path between two nodes. The probability of removing a node in a high-density network that breaks this shortest path is extremely low, because this probability is proportional to $\frac{1}{n}$. We therefore assume that removing a node in a high-density network does not change the length of the shortest paths.

Furthermore, according to (4.11), in a high-density network, the probability that a node located in R_i of A is the i-hop neighbor of A, is approximately 1. That implies that the i-hop communication range of node A is approximately ir in a high-density network, which can be represented as:

$$\lim_{n\to\infty} (i\text{-hop communication range of } A) = ir. \tag{4.12}$$

Therefore, in this dissertation we can apply ir as our approximate i-hop communication range of node A under the assumption of a high-density network which is at least k-connected. The accuracy of this approximation depends highly on the actual network density. The accuracy increases when the network density grows. This will also be confirmed by the experiments in Section 4.5.

Mean Multi-hop Node Degree of Circular Structured Networks

For a circular structured network, we denote the number of nodes that can be reached in the exactly i hops and not in fewer hops, as D_i^c. These nodes are located in the annulus R_i with outer circle radius ir and inner circle radius $(i-1)r$. The area of the annulus can be obtained as the difference of the areas of the two circles. The expected value of D_i^c for a high-density network can be derived from:

$$\lim_{n\to\infty} E[D_i^c] = \begin{cases} 1, & i = 0, \\ n\Big(\pi(ir)^2 - \pi\Big((i-1)r\Big)^2\Big) = (2i-1)n\pi r^2, & i > 0. \end{cases} \tag{4.13}$$

The expected total number of reachable nodes in i hops is $N_i^c = \sum_{j=0}^{i} D_j^c$, which for a high-density network results in:

$$\begin{aligned} \lim_{n\to\infty} E[N_i^c] &= \lim_{n\to\infty} E[\sum_{j=0}^{i} D_j^c] = \lim_{n\to\infty} \sum_{j=0}^{i} E[D_j^c] = \sum_{j=0}^{i} \lim_{n\to\infty} E[D_j^c] \\ &= 1 + \sum_{j=1}^{i} (2j-1)n\pi r^2 = 1 + n\pi r^2 \sum_{j=1}^{i} (2j-1) \\ &= 1 + n\pi r^2 \Big(2\sum_{j=1}^{i} j - i\Big) = 1 + n\pi r^2 i^2. \end{aligned} \tag{4.14}$$

4.2 Cost Functions

To evaluate the performance of our protocol, we observe the traffic load of every node. Traffic load is not only an indicator to bandwidth usage of the network, but also is one of the major sources of battery usage [44]. We aim to develop

an expression for the total cost of transmissions in bits per second per node. In Section 4.2.1, we introduce the assumptions and parameters used in modeling. In Section 4.2.2, we deduce the general cost functions. We focus on false positive probability in Section 4.2.3. Finally, we derive the packet size for both advertisement and query packets in Section 4.2.4.

4.2.1 General Assumptions and Related Vital Parameters

In this chapter, we assume that nodes in the network update frequently. When one node initiates an update, the other nodes might follow due to the changes from that node. To simplify the problem here, we assume that nodes do not update upon changes, but with a constant frequency. Consequently, nodes do not send keep-alive messages in this case. Other nodes can notice the disappearance of one node if they do not receive its updates for a while.

There are several vital attenuated Bloom filters parameters used in our model. In general, we assume that each node has the same number of context information types, s. Further, all context types are supposed to be unique, and taken out of an infinitely large set of possible context types. The same width, w, and depth, d, of attenuated Bloom filters, and the same b hash functions are used throughout the entire network. Moreover, we assume that hash functions are independent and perfectly random. Queries are forwarded by at most d hops, based on the depth of Bloom filters. General notations are listed in the Table 4.1.

Table 4.1: Notation.

General		**Attenuated Bloom filter**	
Notation	Description	Notation	Description
s	number of context information types per node	w	the width of the filter
μ	advertisement(update) rate	d	the depth of the filter
λ	query rate	b	number of hash functions
n	network density (node/m^2)		
r	communication range (m)		

4.2.2 General Functions

We define two types of costs in the network: cost for successful querying, and overhead cost. Cost for successful querying is caused by queries with positive results. Overhead cost is induced by advertisements and false positive queries. The total cost for a node is defined as the sum of the successful querying cost (C_{scq}) and overhead cost (C_{ovh}). Overhead cost is the sum of advertisement cost (C_{adc}) and false positive cost (C_{fp}).

$$C = C_{scq} + C_{ovh}, \tag{4.15}$$

$$C_{ovh} = C_{adv} + C_{fp}. \tag{4.16}$$

Cost for successful querying is the same for different kinds of discovery mechanism. Therefore, in our analysis, we focus on the overhead cost. We assume that advertisements are broadcasted periodically at a constant rate. Therefore the advertisement cost can be defined as:

$$C_{adv} = \mu \cdot adpack, \tag{4.17}$$

where μ is the advertisement (update) rate, and *adpack* is the advertisement packet size.

The false positive cost, C_{fp}, represents the transmission cost for false positive queries incurred by a query initiated by the node under consideration. Transmission of such query messages can take place on all links up to d hops away from the node under consideration. Thus, we can denote the false positive cost as:

$$C_{fp} = \lambda \cdot \sum_{i=1}^{d} cost_{fp,i}, \tag{4.18}$$

where we assume queries are performed at rate λ per second per node. The $cost_{fp,i}$ denotes the total cost of all false positive queries that are transmitted from nodes of the $(i-1)$th hop to nodes of the ith hop.

In order to obtain the false positive query cost, we have to count the maximum possible number of query transmissions between the nodes at the $(i-1)$th and ith hop, $nTr_{fp,i}$. A false positive transmission with packet size *qpack* is, however, only sent when the attenuated Bloom filter of a node at the $(i-1)$th hop gives a false positive in layer $(d-i)$ due to context information that is kept beyond the $(i-1)$th

hop, i.e., that is stored in nodes ranging from the ith till the dth hop. We define $P_{fp,j}$ as the probability of a false positive occurring in layer j. Therefore, this false positive will occur with probability $P_{fp,d-i}$. Note that a false positive in the layers $j < d-i$ automatically implies that there is a false positive in layer $(d-i)$, because of the duplication mechanism we defined (see Section 3.3.3). Thus, false positives in the layers $j < d-i$ do not have to be taken into account. A false positive in a layer $j > d-i$ does not result in the transmission of a query, because the final destination of that query would be more than d hops away from the node that initiated the query. Finally, note that the query sent to the ith hop node will reach that node with certainty, since the false positive in layer $(d-i)$ of the node at hop $(i-1)$ will also appear in layer $(i-1)$ of the node that initiated the query. The resulting false positive query cost from the $(i-1)$th to the ith hop can be given as:

$$cost_{fp,i} = P_{fp,d-i} \cdot nTr_{fp,i} \cdot qpack. \tag{4.19}$$

The maximum number of possible transmissions between nodes $(i-1)$ hops and i hops away, $nTr_{fp,i}$, equals the number of *newly* reached nodes in the $(i-1)$th hop times the average number of direct neighbors, which have not been reached yet. We can calculate these numbers by using the node degree D_i, which was derived in Section 4.1. For $i = 1$, this is:

$$nTr_{fp,1} = D_1. \tag{4.20}$$

For $i > 1$, we have to take into account that one of the neighbors is not a new node, but that it is the neighbor that sent the query packet. In fact, nodes that are receiving a query for the second time, will discard the query message (see Table 3.5 for pseudo-code of parallel querying). Therefore, each query message will be forwarded to at most $(D_1 - 1)$ nodes, which have not received the message before. Note that we assume that the queried context type information is not present in the network. Therefore, independently of the query method (parallel or sequential), the query message will pass through all possible paths. We assume the use of the parallel query method. The number of query transmissions done by nodes $(i-1)$ hops away can thus be counted as:

$$nTr_{fp,i} = D_{i-1} \cdot (D_1 - 1) \qquad (i > 1). \tag{4.21}$$

4.2.3 False Positive Probability

Before we introduce the false positive probability, we first define the number of context information types reachable in the jth hop ($j \geq 0$) for both the grid and circular structures as:

$$x_j = s \cdot N_j. \tag{4.22}$$

We assume that all nodes contain the same number of context type information, s. N_j represents the node degree of the jth hop. For grid structures, we can obtain the exact value (see Section 4.1.2), while we can only obtain an average value for the circular structures (see Section 4.1.2). The second assumption is that all context information types that are reachable within j are represented in the jth layer. A context information type that is represented in a certain layer will also be represented in all lower layers. This holds if the context information of a node is duplicated in all layers of its advertised attenuated Bloom filter, as was described in Section 3.3.3. Even if duplication would not be applied, the same context information will often be present in the lower layers, because in a high-density network a context information type at i hops away can probably also be reached via an alternative path of $j > i$ hops length.

At this point, it is worth mentioning again that we made an important assumption regarding the hash functions. The hash functions are perfectly random, i.e., each bit has an equal probability to be set, independently of the combination of used hash functions. As a consequence, two different context types can in theory be set by the same bits. The probability that such an incidence occurs, however, is in general negligibly small when b becomes sufficiently large. It is also possible that the same bit is set more than once. Hence, the number of bits that represent one context information type can also be smaller than the number of used hash functions, b.

It is probably more efficient to assign specific hash functions to context information types when the total amount of context information in this world is of the same order of magnitude as $b \cdot x_j$ or w. It is safe to assume, however, that in reality, the total amount of context information types is much larger than $b \cdot x_j$ or w. In that case, the use of random hash functions is probably the most efficient and practical way to represent context information. We therefore assert that the assumption of random hash functions is quite realistic.

The assumption of random hash functions enables us to derive the probability of generating a false positive in layer j as a result of a query, $P_{fp,j}$, in a relatively straightforward way. Let the random variable m_j denote the number of bits that are actually set in the layer j. Clearly, $1 \leq m_j \leq \min(b \cdot x_j, w)$, where x_j is the number of context information types represented in layer j. We can obtain $P_{fp,j}$ by using the probabilities conditional on the number of bits set:

$$P_{fp,j} = \sum_{k=1}^{\min(b \cdot x_j, w)} P\{\text{false positive} | m_j = k\} \cdot P\{m_j = k\}. \tag{4.23}$$

Given that the attenuated Bloom filter does not represent the correct context information, we detect a false positive when the hashed context type from the query has the same bits set as those in the advertised attenuated Bloom filter. Because we assume that the hash functions are random for each context information type, the probability that the same bit is set by both (different) hash functions is $\frac{k}{w}$. The probability of a false positive, i.e., the probability that the same b bits are set in both single filters is thus:

$$P\{\text{false positive} | m_j = k\} = \left(\frac{k}{w}\right)^b. \tag{4.24}$$

The probability that exactly k bits are set in the Bloom filter, given that a bit is set $b \cdot x_j$ times, is the quotient of

- the total number of ways to set exactly k different bits out of w bits after setting a bit $b \cdot x_j$ times
- the total number of ways to set bits $b \cdot x_j$ times, given w possible bit positions;

The denominator is equivalent to choosing $b \cdot x_j$ times an element out of w, with replacing the element each time. This equals $w^{b \cdot x_j}$.

It is more complicated to derive the numerator. The total number of ways to set exactly k bits out of w bits after $b \cdot x_j$ times is the product of

- the number of ways to partition $b \cdot x_j$ elements into k non-empty groups;
- the number of permutations of k elements out of a total set of w elements;

In combinatorics, the number of ways to partition a set of n elements into k non-empty subsets is determined by the Stirling number of the second kind, $S(n,k)$, in this case:

$$S(b \cdot x_j, k) = \frac{1}{k!} \cdot \sum_{l=1}^{k} (-1)^{k-l} \cdot \binom{k}{l} \cdot l^{b \cdot x_j}. \tag{4.25}$$

The k subsets in which the $b \cdot x_j$ elements are partitioned have unique identities, e.g., subset 1, subset 2, ..., subset k. Therefore, when in the second step, k elements are selected from a set of w elements, the order of selection must be taken into account. The second factor is thus a permutation rather than a combination, and it can be denoted as $\frac{w!}{(w-k)!}$. Combining all the steps, we derive $P\{m_j = k\}$ as:

$$P\{m_j = k\} = \frac{S(b \cdot x_j, k) \cdot \frac{w!}{(w-k)!}}{w^{b \cdot x_j}}, \tag{4.26}$$

This yields the following equation for the false positive probability in layer j:

$$\begin{aligned} P_{fp,j} &= \sum_{k=1}^{\min(b \cdot x_j, w)} \left(\frac{k}{w}\right)^b \cdot \frac{S(b \cdot x_j, k) \cdot \frac{w!}{(w-k)!}}{w^{b \cdot x_j}} \\ &= \frac{1}{w^{b \cdot (x_j+1)}} \cdot \sum_{k=1}^{\min(b \cdot x_j, w)} k^b \cdot S(b \cdot x_j, k) \cdot \frac{w!}{(w-k)!}. \end{aligned} \tag{4.27}$$

Formula (4.27) is computationally complex, especially for large values of b and w. Therefore, we seek to approximate it. Experiments show that the probability density function of m peaks around its mean when $b \cdot x_j$ is small compared to w. In that case, we can approximate (4.23) by taking the false positive probability at the expected value of m_j, rather than adding the probabilities for all m_j:

$$P_{fp,j} \approx P\{\text{false positive} | m_j = E\{m_j\}\}. \tag{4.28}$$

The expected number of bits set at layer j of the Bloom filter can be expressed by:

$$E\{m_j\} = \left(1 - \left(1 - \frac{1}{w}\right)^{b \cdot x_j}\right) \cdot w. \tag{4.29}$$

Since $(1 - 1/w)^{b \cdot x_j}$ can be approximated by $e^{-b \cdot x_j / w}$ when $b \cdot x_j$ is small compared to w [8], we can substitute (4.24) and further approximate the false positive probability as follows:

$$P_{fp,j} \approx (1 - e^{-b \cdot x_j / w})^b. \tag{4.30}$$

Note that in literature (e.g., [6, 8, 3, 51, 61]), the first approximation step (as expressed in (4.28)) is sometimes presented as being exact. This is not the case, as we have shown in [52, 50]. In order to evaluate the accuracy of the approximation, we do some numerical tests for realistic parameter values. We set $j = 2$ and $s = 1$, so that $x_j = 13$ when considering a grid network (4.3). This implies that 13 context information types are represented in the Bloom filter. Figure 4.4 shows the exact probability $P_{fp,2}$ (4.27) by solid lines and the approximate value (4.30) by dashed lines. The probability $P_{fp,2}$ is shown as a function of w for several values of b. The false positive probability $P_{fp,2}$ decreases with w. The exact false positive probability is slightly higher than the approximate one, especially for low w. However, the difference between the exact and approximate $P_{fp,2}$ is getting smaller when w increases.

The relative difference between the exact and approximate values is shown in Figure 4.5. We use $P_{fp,2}$ to denote the exact value and $P^*_{fp,2}$ to denote the approximate results. Figure 4.5 depicts the relative inaccuracy of the approximation, P_R, as function of w, defined as:

$$P_R = (P_{fp,2} - P^*_{fp,2})/P_{fp,2} \times 100\%. \tag{4.31}$$

For small w, the relative inaccuracy increases with w until a maximum relative inaccuracy is reached. Afterwards, the relative inaccuracy decreases with w. The value of w for which the relative inaccuracy is maximal varies with $b \cdot x_j$, but in all cases the maximum is reached when w is still smaller than $b \cdot x_j$. When w is larger than $b \cdot x_j$, the relative inaccuracy of the approximation becomes smaller. In the optimal situation, we are seeking for a reasonable size of attenuated Bloom filters (w and d) with a certain capacity (b and x_j) that causes few false positives without generating large packets to be advertised. In most cases, it holds that $w > b \cdot x_j$, especially for small j. In any case, the relative inaccuracy appears to be small. We therefore assert that (4.30) can be used to estimate the minimal overhead network cost in our model.

4.2.4 Packet Size

To complete the performance model, we have to specify the packet sizes for advertising and querying. We assume that `Ahoy` runs on top of UDP. The advertisements

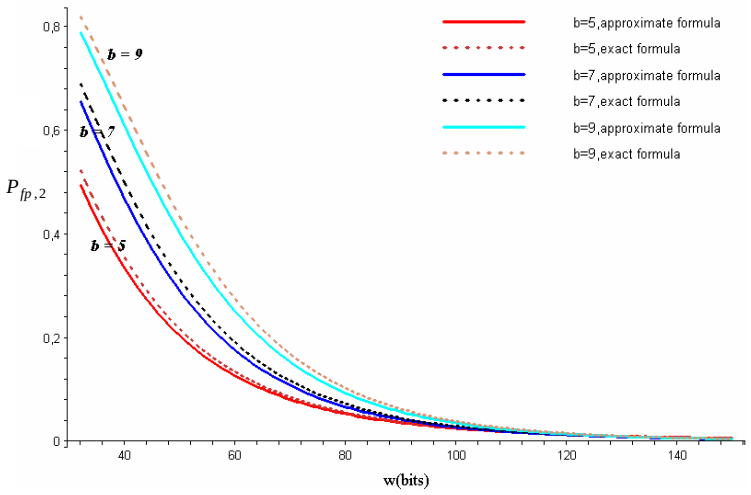


Figure 4.4: Comparison between the exact and approximate false positive probabilities for 3 values of $b(= 5, 7, \text{and } 9)$.

and queries have advertisement (AD) headers and query (Q) headers, respectively . Besides these headers, the headers of the UDP, IP, and MAC layer will be attached. The packet sizes of advertisements (*adpack*) and queries (*qpack*) are defined as follows:

$$adpack = header_{MAC} + header_{IP} + header_{UDP} + header_{AD} + w \times d, \tag{4.32}$$

$$qpack = header_{MAC} + header_{IP} + header_{UDP} + header_{Q} + w. \tag{4.33}$$

The size of the advertisement packet depends on both w and d, because a complete attenuated Bloom filter has to be transmitted. The possibility to apply compression to the filter is beyond the scope of this thesis. The size of the query packet only depends on w, since we assume that queries are sent in the format of Bloom filters.

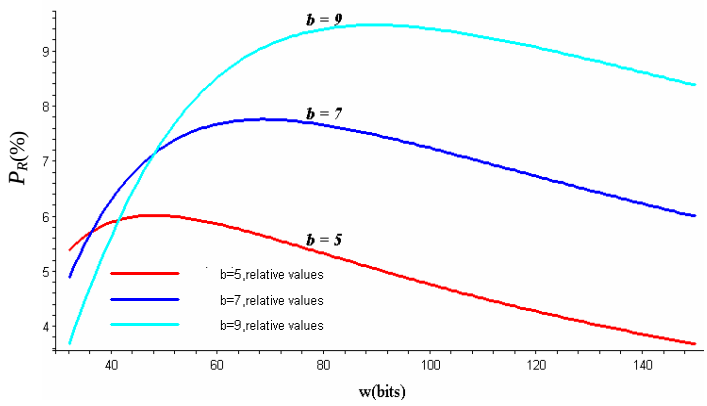


Figure 4.5: Relative inaccuracy of false positive probability for increasing ABF width w and fixed value for $b(= 5, 7, \text{and } 9)$.

4.3 Analysis of two Reference Protocols

We evaluate the performance of `Ahoy` by comparing it with two reference discovery protocols: a so-called proactive protocol and a reactive protocol.

The proactive protocol floods all network nodes within d hops with a complete description of all context information. Nodes therefore have complete knowledge about context information in the network, which implies that nodes can directly send queries to nodes that contain the required information. The advertisement cost is the main concern in this situation. We assume that each context information type can be presented in c bits, and that each node broadcasts its advertisement within a $(d-1)$ hop range. Hence, we can derive the discovery cost for the proactive protocol, $cost_{proact}$, as:

$$cost_{proact} = \mu \cdot N_{d-1} \cdot (header_{MAC} + header_{IP} + header_{UDP} + header_{AD} + s \cdot c). \quad (4.34)$$

In the case of *the reactive protocol*, nodes do not advertise context information types in advance. An incoming query is forwarded to all neighbors. Nodes do not

have any idea about context information in the network. The queries are spreading throughout the whole network, up to d hops from the initiator. There is no way to stop forwarding queries without generating more traffic in the network, even though the querying node has already received an answer. The cost for querying is counted as the cost for broadcasting queries to all nodes in the network. All nodes up to d hops away will receive such a query, apart from the node that initiated the query. The number of transmissions is therefore equal to the total number of nodes within d hops minus 1. The discovery cost for the reactive protocol, $cost_{react}$, can be obtained as:

$$cost_{react} = \lambda \cdot (N_d - 1) \cdot (header_{MAC} + header_{IP} + header_{UDP} + header_Q + c). \quad (4.35)$$

4.4 Experimental Results

We implemented the model described in Section 4.2.2 in Matlab 7.1 to evaluate the performance of `Ahoy`. We focus on the overhead network cost of the proposed system and its alternatives. Therefore, we assume that the context information types that are queried for are not present in the network, i.e., the cost for successful querying, $C_{scq} = 0$.

Experiment 1 is designed to obtain the optimal cost by choosing the proper width of the Bloom filter, w, and the optimal number of hash functions, b, given the depth of the filter, d, advertisement rate, μ, query rate, λ, and number of context information types per node, s. In experiments 2, 3, 4, and 5, we study the impact by varying certain parameters, i.e., the query rate (λ), the query range (d), the information density (s), and the network density (n).

As we introduced in Section 4.1.1, we use two network structures: grid and circular-structured network. Experiments 1 and 2 are done on both network structures. Compared to the communication range, a grid-structured network has a predetermined network density, which restricts the possibility of observing `Ahoy`'s performance related to different network densities. Compared to the grid structure, the circular structure has the advantage that the number of neighbors of a node is

not fixed. Therefore, the three other experiments (experiment 3, 4, and 5) are only being done for the circular structure.

For a fair comparison between the grid and circular structure in experiments 1 and 2, we use a node density at which the circular and grid structure resemble each other the most. This is the case when the i-hop node degrees in both grids are equal, $N_i^g = N_i^c$:

$$1 + 2i(i+1) = 1 + n\pi r^2 i^2 \tag{4.36}$$

$$\Rightarrow n\pi r^2 = \frac{2i(i+1)}{i^2} = 2 + \frac{2}{i} \qquad (i > 0). \tag{4.37}$$

If i is large, the circular structure corresponds with the grid structure for:

$$n\pi r^2 \approx 2. \tag{4.38}$$

In all the experiments below, we assume that each context information type can be represented in 32 bits, i.e., $c = 32$ bits. The sizes of headers are assumed as follows $header_{MAC} = 256$ bits [38]; $header_{IP} = 320$ bits (assuming the use of IPv6 [43]); $header_{UDP} = 64$ bits [21]; $header_{AD} = 32$ bits; $header_Q = 192$ bits.

4.4.1 Basic experiments

Experiment 1

In this experiment, we aim to obtain the optimal network cost, its corresponding size of the ABF (w), and the number of hash functions (b) for each query range d, given fixed advertisement and query rates μ and λ and number of context type information per node s. The query range d refers to the number of hops that a query travels. This number should be equal to the depth of the ABF, which therefore is also denoted by d. Based on our derivation in Section 4.2, the network cost is a function of the width of the filter w and the number of hash functions b, given certain values for d, λ, and s. Due to the discrete values of w and b, we are able to calculate the network cost of each pair of w (from 1 to 2^{16} bits) and b (from 1 to w/x_i). We thus obtain the minimal network cost without using an optimization algorithm. In this experiment, we set an advertisement rate of $\mu = 0.1$ and a query rate of $\lambda = 0.1$, and one context information type per node $s = 1$. The filter depth d varies from 3 to 10.

In Table 4.2 and Table 4.3, we show the minimum network costs in bit/s for the grid and circular network structure respectively. We show the cost for `Ahoy`, the proactive protocol, and the reactive protocol (using the same parameter settings). From Table 4.2 and Table 4.3, we conclude that for each d, optimal values for the width, w, and number of hash functions, b, lead to minimum network costs that are lower than those for the proactive and reactive protocols. `Ahoy` reduces network traffic due to overhead with about a factor 10 to 20 compared to the proactive and reactive protocol respectively. `Ahoy` therefore improves the network traffic enormously when nodes advertise and query context information every 10 seconds.

The last columns of Table 4.2 and Table 4.3 show the maximum number of context information types that are to be represented in one Bloom filter (x_d given d). The density of the circular network is chosen such that the node degree is equal to the network density of grid networks for large distances. For larger d, the maximum number of context information types is therefore more or less equal for both network structures. However, for small d, less context type information is stored for the circular structured network than for the grid network. As a result, a larger attenuated Bloom filter is needed for the circular structured network, and slightly more network cost is generated than is indicated in the fourth column of Table 4.3.

Table 4.2: Optimal ABF cost compared with the proactive and the reactive protocols (grid structure).

d	w (bits)	b	Optimal `Ahoy` cost (bit/s)	Proactive protocol (bit/s)	reactive protocol (bit/s)	Maximum number of context information types in ABF
3	96	5	107	915	2074	13
4	160	5	155	1760	3456	25
5	256	5	230	2886	5184	41
6	352	5	341	4294	7528	61
7	448	5	495	5984	9677	85
8	608	5	701	7955	12442	113
9	736	5	967	10208	15552	145
10	928	5	1304	12742	19008	181

Table 4.3: Optimal ABF cost compared with the proactive and the reactive protocols (circular structure).

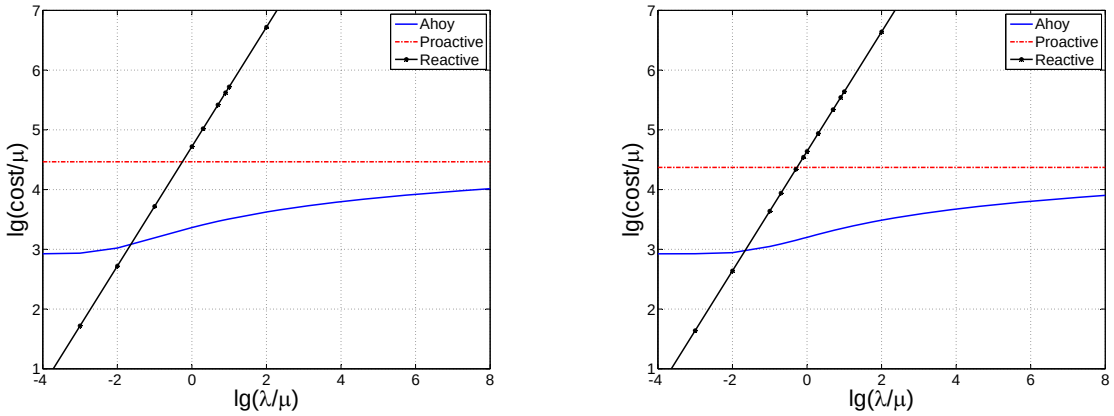
d	w (bits)	b	Optimal `Ahoy` cost (bit/s)	Proactive protocol (bit/s)	Reactive protocol (bit/s)	Maximum number of context information types in ABF
3	64	5	89	634	1555	9
4	96	4	116	1338	2765	19
5	128	4	158	2323	4320	33
6	192	4	222	3590	6221	51
7	256	4	312	5139	8467	73
8	320	4	435	6970	11059	99
9	416	4	597	9082	13997	129
10	512	4	804	11475	17280	163

Experiment 2

In experiment 1, we showed that simple and efficient Bloom filters can reduce the network load significantly. However, so far we only used one combination of update frequency μ, and query rate λ. Redundant traffic due to false positives may lead to problems when the query rate increases. We expect that there exists a (high) value for λ at which the costs due to false positives outweighs the benefits of using Bloom filters. In contrast, if the query rate is very low, and only few queries are sent through the network, there is no benefit to broadcast the context information. The reactive protocol might perform better in that case. In this experiment, we study for which range of μ and λ `Ahoy` is optimal. This enables us to achieve a favorable network cost by using the most suitable protocol under all circumstances.

As a reference, we set μ equal to 0.1 /sec, and change the value of λ to tune the ratio λ/μ. First of all, we study the case in which d is equal to 5. For both the grid and circular structures, we analyze the situation in which each node has only one unique context information type ($s = 1$). The results are shown in Figure 4.6(a) and Figure 4.6(b), respectively. They clearly demonstrate the range of λ/μ where `Ahoy` generates the least of overhead cost. To show the impact of the ratio λ/μ, overhead cost presented in the y axis of the both figures is also divided by μ accordingly. Moreover, both axes have a *logarithmic scale*. Both figures show that for medium and large values of λ/μ, the `Ahoy` context discovery algorithm generates less traffic than the traditional proactive and reactive algorithms. For both networks, network cost is lower for `Ahoy` than for the reactive protocol when λ/μ is larger than 0.1, i.e., if queries are generated at least once per 10 advertisement periods. The cost difference between the two algorithm increases exponentially with λ/μ. On the other hand, the difference between `Ahoy` and the proactive algorithm decreases with λ/μ. In the range shown by the figures, however, `Ahoy` always performs better. If we would extrapolate both cost curves, we would find a very high value of λ/μ (much larger than 10^8) beyond which the proactive protocol is more cost effective.

We extended the experiments by varying d from 3 to 10. Similar results are obtained as for $d = 5$, as are shown in Figure A.1 in Appendix A. In the range of medium and high values of λ/μ `Ahoy` generates less network traffic than the two traditional algorithms. When d increases, the amount of traffic is increasing. The relative increases are more or less the same for all protocols. For $d = 10$, `Ahoy`



(a) Grid structure

(b) Circular structure

Figure 4.6: Overhead costs generated by `Ahoy`, the proactive and the reactive protocols while varying λ/μ while $d = 5$ and $s = 1$.

outperforms the reactive protocol when λ/μ is larger than about 0.01. For large λ, the relative increase in cost is somewhat larger for `Ahoy` than for the proactive protocol. Probably this is the result of extra traffic due to false positives. Note however, that for small λ, the relative cost difference between the proactive protocol and `Ahoy` increases rather than decreases with d. The differences between $d = 3$ and $d = 10$ are however very small. In all cases, the range for which `Ahoy` outperforms both other protocols is quite comparable with that for $d = 5$.

We also increase the context information density s, and assume $s = 4$. The network then requires larger Bloom filters to contain more information. Figure A.2 in Appendix A shows the results. Similarly to the situation where $s = 1$, there exists a large range in λ/μ for which `Ahoy` generates less network cost than the traditional protocols. As might be expected, this range becomes smaller when the information density s increases from 1 to 4 (see Figure A.1 and A.2). When $d = 3$, `Ahoy` outperforms the traditional protocols for λ/μ between 0.1 and at least 10^8. When $d = 10$, the range is from 0.1 to 1000 in the grid-structured network, and 0.01 to 4000 in the circular-structured network.

The results show that `Ahoy` has a better performance in terms of generating less traffic in most realistic situations, i.e., in the situation where the frequencies of

sending out advertisements and queries are close to each other. The exact range can be calculated from our model, and varies slightly with different query range d and information density s.

4.4.2 Extensive experiments

The previous two experiments have shown that `Ahoy` performs quite well. In the following experiments we further evaluate the performance of the different protocols as function of the discovery range d, and the information density s. We also study the impact of node density n on the network cost. In the following three experiments, we only use circular structured networks. There are two reasons for this choice. First, a circular structured network resembles reality more closely than a grid structured network. Secondly, the network density can be varied in circular structured networks, whereas a grid structured network only has one fixed network density with respect to the communication range. Also, from now on, we only show the overhead cost on a linear scale, because in practice we are mainly interested in absolute cost differences between protocols.

Experiment 3

With a larger discovery range d, more context information types are available for the querying node. On the other hand, the size of the attenuated Bloom filters will also increase. In this set of experiments, we study the impact of d on the overhead network cost.

We vary the depth of the Bloom filter, d, from 3 to 10, and compare the performance for different values of s and λ (fixed $\mu = 0.1$). Figure 4.7 displays the network costs as function of d given that $s = 1$ and $\lambda = 0.1$. `Ahoy` generates less network traffic than both the proactive and reactive algorithms. The absolute difference in network costs increases with d, i.e., the cost increases much faster for the other two alternatives than for `Ahoy`. The cost for `Ahoy` increases slightly with d. When $d = 3$, `Ahoy` generates 545 bit/s less than the proactive algorithm and 1466 bit/s less than the reactive algorithm. When $d = 10$, `Ahoy` generates 10671 bit/s less than the proactive algorithm and 16476 bit/s less than the reactive algorithm. However, it should be noted that the relative increases in cost are quite comparable for the different protocols, as was shown in experiment 2.

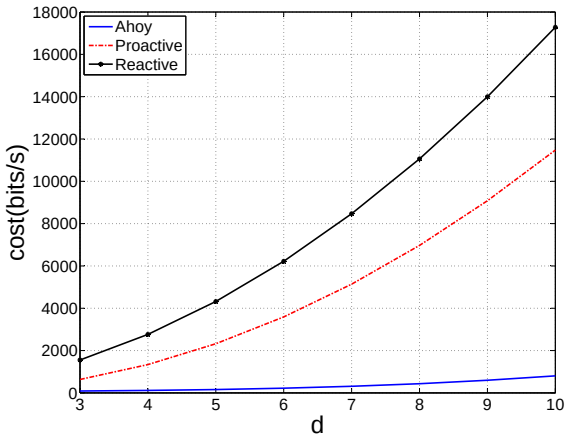


Figure 4.7: Overhead cost of `Ahoy`, the proactive and reactive protocols while varying the ABF depth, d and setting $s = 1$, $\lambda = 0.1$, $\mu = 0.1$.

Next, we evaluate more general cases with λ equals to 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, and 1000, respectively. The experiments were done for both $s = 1$ and $s = 4$, as is shown in Figure A.3 and Figure A.4 in Appendix A respectively. The results show that, in general, the `Ahoy` algorithm generates less network traffic than both the proactive and reactive protocol. The performance of `Ahoy` compared to the other protocols is strongly related with the query rate (λ) and the number of context type information within the query range (this depends on both d and s). When the number of context information types is very small, the network cost is relatively large in `Ahoy` compared to the reactive protocol. However, the reactive protocol only outperforms `Ahoy` for very small ($\lambda = 0.0001$ and $\lambda = 0.001$). For all other query rates, `Ahoy` has less network cost than the reactive protocol. In contrast, when the query rate is quite high, the cost of `Ahoy` increases significantly and may be more than the proactive protocol. However, this is only the case when $s = 4$, $\lambda = 1000$, and $d > 9$. For all other cases, `Ahoy` outperforms the proactive protocol. Note that for high λ the relative increase in cost with d is lower for the proactive protocol than for `Ahoy` (as has already been shown in experiment 2). Therefore, the proactive protocol might score somewhat better if we would increase the range, d.

The results from Experiment 3 confirm the results of Experiment 2. `Ahoy` can save network costs compared to the two other alternatives. There exists an upper

and lower bound of d for the optimal performance of `Ahoy`, which depends on the query rate (given that the advertisement rate is fixed), and the number of context information types within the query range. The latter depends on the query range d and the number of context information types per node s.

Experiment 4

The previous experiments show that the number of context information types per node s also has an influence on the network cost. In this set of experiments, we investigate this in detail. We examine the network costs as function of s. For different values of d and λ (with fixed μ), we change s from 1 to 6, i.e., the number of context sources within the query range increases from 51 to 306. We start the study again with a realistic situation where $d = 5$, $\lambda = 0.1$, and $\mu = 0.1$. The result is shown in Figure 4.8. `Ahoy` generates the least network traffic among three algorithms as expected. The network costs of `Ahoy` and the proactive algorithm exhibit a slight increase with increasing s. Note that the cost of the reactive protocol stays the same, because it is only related to the number of node in range, the query rate, and the query packet size.

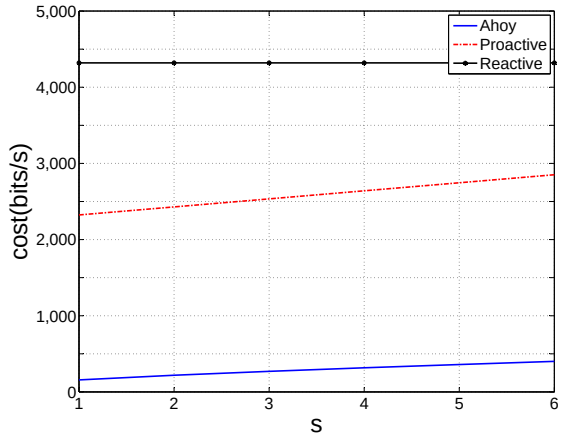


Figure 4.8: Impact of the context type density, s, on the overhead cost of `Ahoy`, the proactive and the reactive protocols, while $d = 5$, $\lambda = 0.1$, $\mu = 0.1$.

Next, we model the three algorithms for λ equals to 0.001, 0.01, 0.1, 1, and 10, respectively, and for $d = 3$, $d = 5$, and $d = 10$. The results are shown in Figure A.5 to A.7 in Appendix A. In most cases, `Ahoy` generates much less network costs than the other two algorithms. There are only a few exceptional situations in which `Ahoy` generates more network traffic than the reactive algorithm. This occurs when d =3, or 5, and $\lambda = 0.001$ (see Figure A.5(a) and Figure A.6(a)), and when $d = 10$, $\lambda = 0.001$, and $s > 3$ (see Figure A.7(b)). The figures show that `Ahoy` generates less network traffic than the proactive algorithm in all current experiment settings. However, for $d = 10$ and $\lambda = 10$, we can expect that the network cost of `Ahoy` will exceed the cost of the proactive algorithm when s is larger than 6, because the cost in `Ahoy` increase faster with s than those in the proactive protocol (see Figure A.7(g)). These results are consistent with the results from Experiments 2 and 3. Obviously, within a certain boundary, `Ahoy` performs better in terms of network traffic than the alternatives. As mentioned before, the optimal range for `Ahoy` depends on the query rate and the number of context information types within the query range. The figures from experiment 3 show that the effect of s on the network cost is relatively mild compared with that of d, given that the other parameters are the same.

Experiment 5

In the previous experiments, we used a network density of $n\pi r^2 = 2$, i.e., an average node has 2 neighbors in a circular structured network. The previous experiments have shown a relationship between the number of context information types in the query range and the performance of `Ahoy`. Therefore, we anticipate that network densities also influence the network cost directly. In the circular structured model, we can easily vary this parameter. The number of nodes within communication range was varied from 2 to 20, which implies that $n\pi r^2 = 2$ to 20. The other parameters were set as follows: $\lambda = 0.1$; $\mu = 0.1$; $d = \{3, 5, 7, 10\}$. We did two sets of experiments with $s = 1$ and $s = 4$.

We present the relation between network cost and network density when $s = 1$ in Figure 4.9. The results when $s = 4$ are similar as when $s = 1$, and are presented in Figure A.8 in Appendix A. The results show that, in general, the costs increase almost linearly with the number of direct neighbors. The relative increase is more or less the same for all protocols. Because `Ahoy` performs better than the other two

protocols, the network cost of Ahoy increase slower in absolute terms. Ahoy therefore also performs significantly better when densities are very high.

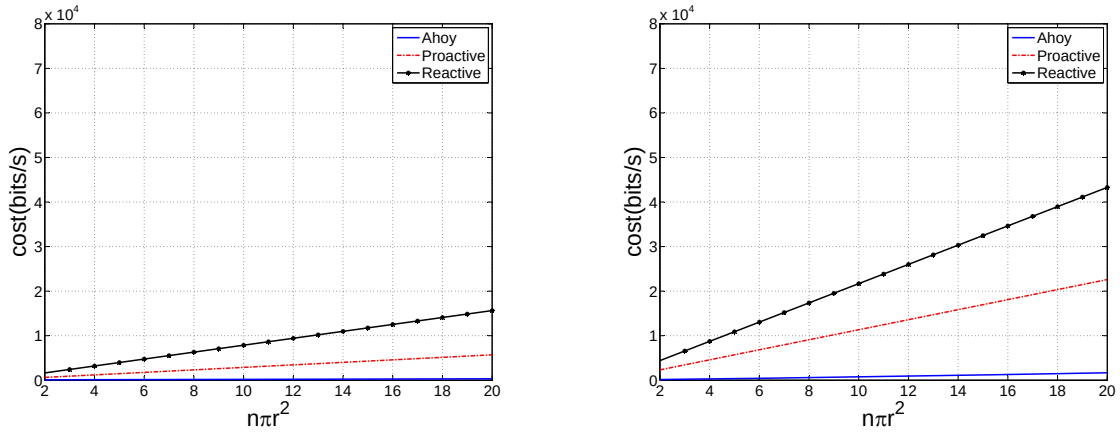
In this experiment, we also confirm the boundaries within which Ahoy performs better than the other two alternatives. As was shown previously, the boundaries are related to the query rate and the amount of context information within the query range. Beyond the boundaries, Ahoy generates more network traffic than the alternatives. At the lower boundary, Ahoy and the reactive protocol perform equally well. This is the case, for example, when $\lambda = 0.003$, $s = 4$, $d = 3$, and the number of neighbors is 4. When the density decreases, querying by the reactive protocol becomes relatively cheap. For $\lambda = 0.003$, $s = 4$, and $d = 3$, the reactive protocol therefore outperforms Ahoy when each node has on average fewer than 4 neighbors. This is illustrated in Figure 4.10(a). Contrarily, in a dense network with frequent queries (upper boundary), Ahoy generates more network traffic than the proactive protocol. For example, when $\lambda = 100$, $s = 4$, and $d = 10$, the proactive protocol outperforms Ahoy when each node has at least 4 neighbors. This is illustrated in Figure 4.10(c). The cost for the reactive protocol fall off this chart and are therefore shown separately in Figure 4.10(b).

4.4.3 Summary

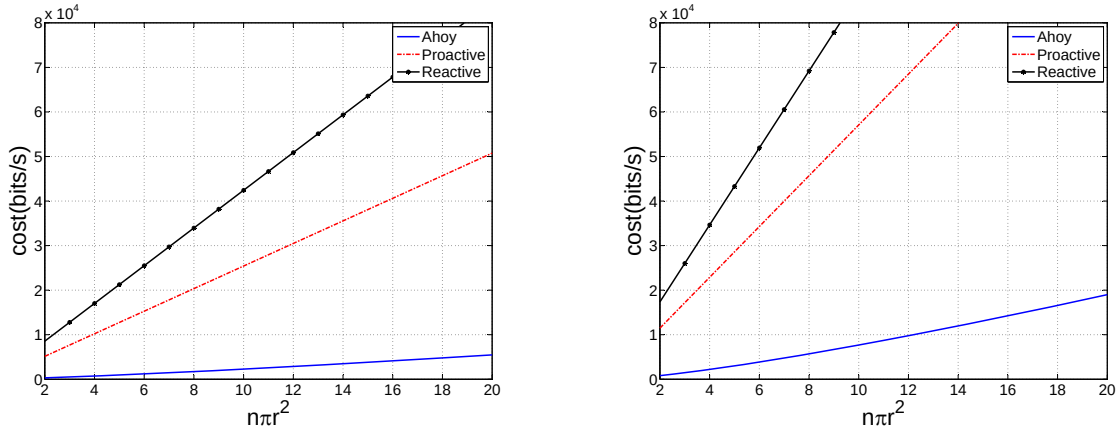
In this section, we used an analytical model to investigate the performance of the Ahoy protocol in terms of generated traffic. We studied the performance of Ahoy in two different network structures: a simple grid network and a circular network which represents a fully distributed ad-hoc network. We focussed on the overhead costs. These are the combined costs of advertising and querying in case the requested information is not available. In Ahoy unsuccessful queries are also related to false positives. For both networks we were able to set the optimal parameters for Ahoy in which there is a balanced trade-off between the number of false positives and the size of the Bloom filters.

In a comparison with the proactive and reactive protocols, we find similar results for both network structures. Ahoy performs better than the traditional algorithms within certain boundaries which are associated with the ratio of query and advertisement rates, the query range, the density of context information sources, and the node density. For each setting of network parameters, our model can compute the

boundaries and obtain the optimal ABF settings (including w and b) that generate the minimum network cost. Results demonstrate that for most practical situations, `Ahoy` requires significantly less (up to an order of magnitude) traffic load than the proactive and reactive protocols. This is very beneficial for MANETs, because this indicates that `Ahoy` consumes less bandwidth and battery usages of nodes in the network. As such, it comprises a very effective and efficient protocol for real world applications.

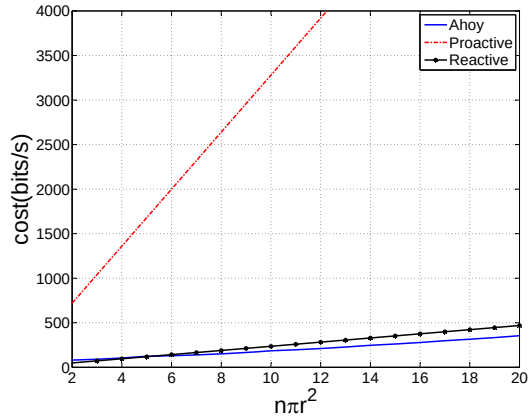


(a) $d = 3$ (b) $d = 5$

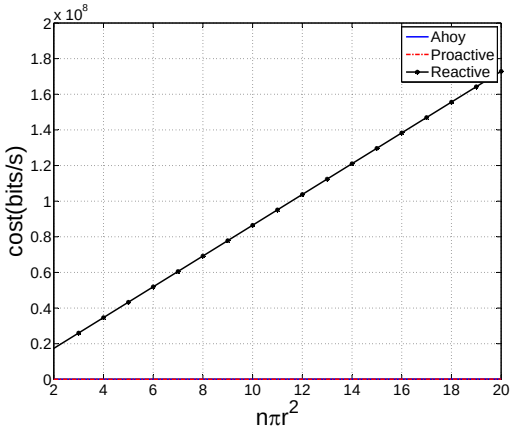


(c) $d = 7$ (d) $d = 10$

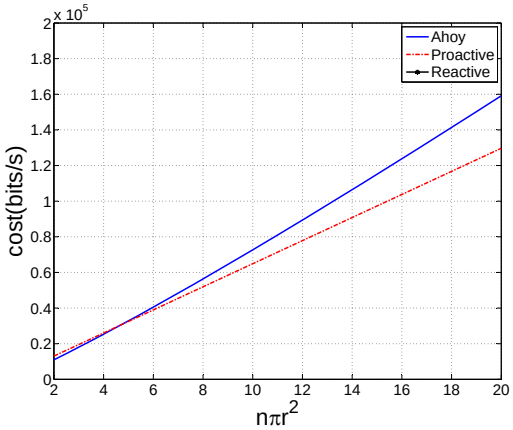
Figure 4.9: Impact of network density, n, on the overhead cost of `Ahoy`, the proactive and the reactive protocols, while $s = 1$, $\lambda = 0.1$, $\mu = 0.1$.



(a) $d = 3$



(b) $d = 10$



(c) $d = 10$

Figure 4.10: Boundary of `Ahoy` in terms of network density while $s = 4$ and $\mu = 0.1$. $\lambda = 0.003$ in the upper panel, and $\lambda = 100$ in the lower panel.

4.5 Model Validation

With the analytical results from Section 4.4 we are able to find the optimal parameter settings for `Ahoy`, and to estimate the network conditions under which `Ahoy` performs optimally, i.e., better than other protocols. In this section, we use a simulation [28] to evaluate the accuracy of our analytical model.

The evaluation is done by comparing the overhead cost observed in a detailed simulation study with our analytical model. We first introduce this simulation in Section 4.5.1. Next, we show that the overhead cost from both models are equivalent to each other in Section 4.5.2. In Section 4.5.3, we address the simulation setup used for comparison. In Section 4.5.4, we present the results from both analytical model and simulations and analyze them. Finally, we conclude the validation in Section 4.5.5.

4.5.1 Brief Introduction to Simulation Model

The simulation model [28] has been implemented in the discrete-event simulator OPNET Modeler [63] version 11.5. A MANET node from the OPNET model library was modified to support our protocol. All nodes utilize the IEEE802.11b standard [40] with a bit rate of 11 Mbps for communication with each other. The discovery protocol is built on top of UDP over IPv4. Note that we utilize IPv4 instead of IPv6 here, because when the simulation model was implemented, OPNET 11.5 was not good enough to support modeling on top of IPv6.

In this section, we validate the analytical study for the grid structured network. For the circular structured networks, the comparison between simulations and analytical results will be covered in detail in Chapter 5. In grid structured networks, we can precisely count the number of transmissions for each hop. We build a network of 61 nodes with every node having 4 direct neighbors in the transmission range. The center node can reach any node in the network within 5 hops. The maximum communication range is 300 meters. To have a good comparison result with our analytical model, it is assumed that nodes beyond 300 meters are not interfered by the transmission, nor will their carrier sense the transmission.

The random generator from OPNET is utilized for all random generations in our model, including both time and text strings. No retransmission mechanism for

broadcast packets has been implemented when collisions occur in `Ahoy`. The messages are unsynchronized, i.e., they are sent at random moments. Random text strings are generated to represent the context information types. Context information types are randomly generated for every simulation run, using different seeds for context exchange and querying. Universal hashing [11] was chosen to hash the context information types.

We investigate the cost of false positives as well as the cost of sending advertisement messages. In this section, the cost is represented by the overhead traffic that is generated by the center node.

4.5.2 Proof of Equivalent Overhead Cost

We compare the overhead network cost from both the analytical and simulation models in the rest of the section. Although we calculate the overhead network cost in two models from different perspectives, in essence, they are equal to each other.

In the analytical model, the overhead cost is the sum of costs caused by packets which are initiated by a querying node. We count all traffic that is related to the query packets initiated by that one node. In the simulation model, the overhead network cost is basically counted as being the number of packets that are transmitted by one (central) node. Some of those packets are not initiated by that node, but are the result of updating changes and forwarding queries that are initiated by other, reachable nodes. In the following we will show that both calculations are equivalent. The simulation overhead cost (C^s_{ovh}) is the sum of the advertisement packets cost (C^s_{adv}) and the false positive query packets cost (C^s_{fp}):

$$C^s_{ovh} = C^s_{adv} + C^s_{fp}. \tag{4.39}$$

The superscript s denotes that the variables refer to results for the simulations.

C^s_{adv} can also be written in the same way as for the analytical model in (4.17). Therefore, we have:

$$C^s_{adv} = C_{adv} = \mu \cdot adpack. \tag{4.40}$$

C^s_{fp} is caused by the false positive queries transmitted by the center node. This includes two types of queries: queries that originate from the center node and queries that originate from other nodes. The queries from the center node will be forwarded at most d hops. The queries from nodes that are i hops away will be propagated at

most $(d-i)$ hops further. C^s_{fp} can be described as the sum of queries forwarded up to d hops away from the center node:

$$C^s_{fp} = \lambda \cdot \sum_{i=1}^{d} cost^s_{fp,i}, \tag{4.41}$$

where $cost^s_{fp,i}$ represents the total false positive cost for queries that are being transmitted i hops further until they reach the nodes at d hops from the center node. Those queries are thus initiated by nodes $(d-i)$ hops away from the center node. They are generated or forwarded if there is a false positive in the layer $(i-1)$ of the ABF of the center node. The queries that are initiated by the center node ($i = d$, because they will be forwarded d hops further) will be transmitted to 4 neighbors. The queries that are initiated in the $4 \cdot (d-i)$ nodes at $(d-i)$ hops from the center node, will be forwarded to 3 neighbors. Therefore, the total false positive cost can be calculated as:

$$cost^s_{fp,i} = \begin{cases} P_{fp,i-1} \cdot 4(d-i) \cdot 3 \cdot qpack, & i < d, \\ P_{fp,d-1} \cdot 1 \cdot 4 \cdot qpack, & i = d. \end{cases} \tag{4.42}$$

If we set $j = d - i + 1$, (4.42) can be rewritten as:

$$cost^s_{fp,d-j+1} = \begin{cases} P_{fp,d-j} \cdot 4(j-1) \cdot 3 \cdot qpack, & j > 1, \\ P_{fp,d-1} \cdot 1 \cdot 4 \cdot qpack, & j = 1. \end{cases} \tag{4.43}$$

Therefore, (4.41) can be rewritten as:

$$C^s_{fp} = \lambda \cdot \sum_{j=1}^{d} cost^s_{fp,d-j+1}, \tag{4.44}$$

which is equivalent to (4.18) from the analytical model. Therefore, we have:

$$C^s_{fp} = C_{fp}. \tag{4.45}$$

By substituting (4.40) and (4.45) into (4.39) and applying (4.15), we obtain:

$$C^s_{ovh} = C_{adv} + C_{fp} = C. \tag{4.46}$$

We conclude that counting all query packets sent out by a single node (which may be initiated by nodes up to $(d-1)$ hops away) is equivalent to counting all query packets which are sent by nodes up to $(d-1)$ hops away, as a result of a query that is generated in a single node. The latter approach is taken in the analytical model, whereas the former approach is taken in the simulation model. The cost functions of the two different models are therefore equivalent to each other.

4.5.3 Comparison setup

Based on the discussion in Section 4.5.2, we configure the simulation model as follows to achieve equivalent settings for the analytical model. We focus on the center node in the simulation, so that we will not observe boundary effects. We want to estimate the overhead costs. Therefore, we assume that the queried context information type is not available in the network, as also assumed in our analytical study. We count the number of packets that are transmitted by the center node, including periodical broadcast packets and query packets. We divide the total number of transmitted bits by the total simulation time (except the first 20 minutes for initialization) to obtain the overhead network cost per node. Since context information types and queries were randomly generated, 100 independent runs, with 10 simulation hours per simulation, were done to calculate 90% confidence intervals of the overhead network cost. The related analytical results are compared with the averages and the confidence intervals from the simulations.

In the simulation experiments, nodes artificially refresh the attenuated Bloom filters and advertise them periodically (without being triggered by the keep-alive mechanism, to be aligned with the analytical model). We assume the advertisement period to be 600 seconds plus a random time that was uniformly chosen between 0 and 10 seconds to reduce collisions. During one advertisement period, 600 queries were sent. Therefore, $\mu = 1/605$ sec and $\lambda = 600/605$ sec on average. These averages are also used in the analytical model.

In this section, we use simulations to evaluate the accuracy of the analytical model based on the approximation (4.30) and the exact false positive probability (4.27). We implement the analytical model with approximate false positive probability (4.30) in Matlab 7.1. Matlab 7.1 cannot compute Stirling numbers of the 2nd kind when the parameter $b \cdot x_j$ exceeds 252. Therefore, we use Maple 9.5 to calculate (4.27), in order to achieve the overhead cost with exact false positive probabilities.

Some basic experiments parameters are set as follows: $header_{MAC} = 256$ bits; $header_{IP} = 160$ bits (assuming the use of IPv4 [66]); $header_{UDP} = 64$ bits; $header_{AD} = 32$ bits; $header_Q = 192$ bits. We assume one context information type per node so that $s = 1$.

4.5.4 Experimental Results

Three experiments have been done to compare the results from the analytical and simulation models. Experiment 1 demonstrates the evaluation of network cost for varying width (w) of the Bloom filters. Experiment 2 verifies the optimal value for w and b given depth d. Finally, experiment 3 compares the network cost for `Ahoy` and the proactive and reactive discovery protocols.

Experiment 1

In this experiment, we observe the network cost for different ABF widths (w). Two sets of experiments have been done: in experiment (a), $d = 3$ and $b = 15$ (see Figure 4.11) and in experiment (b), $d = 5$ and $b = 13$ (see Figure 4.12). In both cases, the value of b corresponds to the optimum value according to the analytical model.

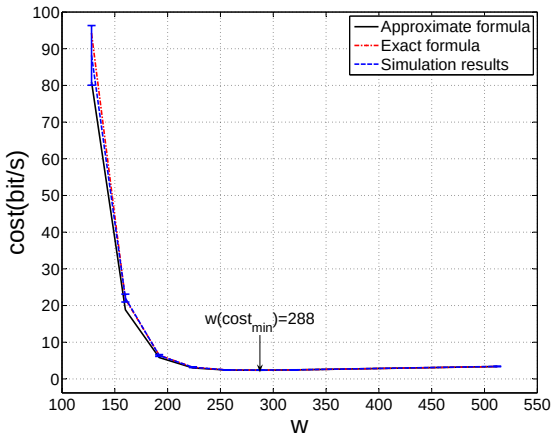


Figure 4.11: Network cost for $d = 3$, $b = 15$.

The figures show that the analytical and simulation results are very close to each other. The simulation confirms the values of w and b that yield a minimum network load. The value of w is the same as the one from the analytical model, which is 288 bits for $d = 3$ and 768 bits for $d = 5$. The approximated results (4.30) and exact results (4.27) are also very similar, especially when $b \cdot x_d$ is smaller than w, i.e., when

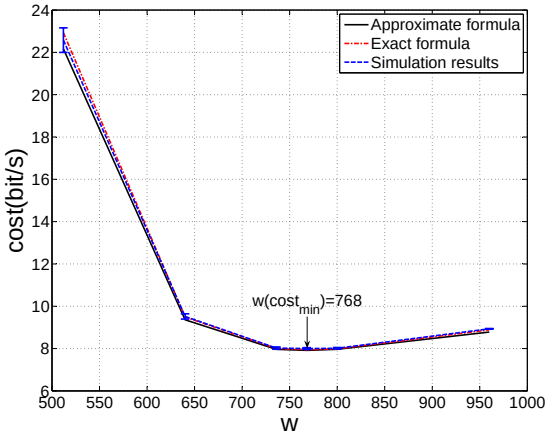


Figure 4.12: Network cost for $d = 5$, $b = 13$.

the false positive probability is small. The results from the exact analytical formula are always within or very close to the 90% confidence interval of the simulation results. Except for $w < 150$, the confidence intervals are very small and almost invisible.

The figures also show that the overhead cost rapidly decreases with an increase of w till it reaches a minimum. Beyond that minimum, the cost increases only slightly with w. Normally we determine the value of w and b in advance (given an expected number of available context information types in the network). We suggest to set w at a slightly larger value than the expected optimum. A larger w only leads to slightly more overhead cost, while a smaller w can lead to much more cost, especially when the optimal value varies due to moving nodes.

Experiment 2

One of the purposes of the analytical models is to estimate the optimal attenuated Bloom filter size and number of hash functions to achieve the minimum network load. We ran a number of simulations with different values for the parameters w and b with d equals 3, 4, and 5, respectively.

The results are shown in Table 4.4. According to the table, the values of optimal values for w and b are exactly the same for the three models. The network costs are

Table 4.4: Optimal cost generated by `Ahoy` and the related value of w and b when $d = 3,\ 4,$ and 5.

	`Ahoy` **Approximate**			`Ahoy` **Exact**			**Simulation Results**		
d	w (bits)	b	costs (bit/s)	w (bits)	b	costs (bit/s)	w (bits)	b	90% confidence interval of costs (bit/s)
3	288	15	2.37	288	15	2.38	288	15	(2.39, 2.40)
4	480	13	4.44	480	13	4.50	480	13	(4.54, 4.61)
5	768	13	7.92	768	13	7.94	768	13	(7.96, 8.04)

slightly different, due to differences in the estimates of the false positive probabilities. For the optimal situation, the approximation is always underestimating the costs, but not more than 1% compared to the exact analytical model. The costs from the analytical models are slightly less than those from the simulation, and also just outside the 90% confidence interval. However, the difference is not more than 1%.

According to experiments 1 and 2, the analytical and simulation models provide very similar results. The approximate and exact analytical models are both quite accurate in achieving the optimal parameters. The exact model achieves more accurate outcomes, especially when the costs are high. However, the approximate model can still provide very good estimates, and is more suitable for calculations in large networks. Furthermore, the simulation model is more suitable to test more complicated situations, such as the dynamic networks, which will be discussed in Chapter 5.

Experiment 3

As discussed in Section 4.4.1, the performance of attenuated Bloom filters is highly dependent on the ratio between query and advertisement rate. For a large range of realistic values, `Ahoy` achieves lower network costs than the alternative solutions. In this experiment, we verify the results from Section 4.3 by simulations.

We assume that each context information type can be represented in 32 bits, i.e., $c = 32$ bits. We fix μ as 1/(605 sec) and vary the value of λ from 10^{-3}/(605 sec) to 10^{7}/(605 sec). The experiment has been done for values of $d = 3$. Figure 4.13

demonstrates the results. **Note that the scale of the figure is such that simulation and analytical results of `Ahoy` cannot be distinguished from each other.** The experiment results show that the simulation model gives almost exactly the same results as the analytical models. The same conclusions can thus be drawn as in Section 4.4.1; `Ahoy` consumes less network traffic than the two conventional approaches within a wide range of query rates.

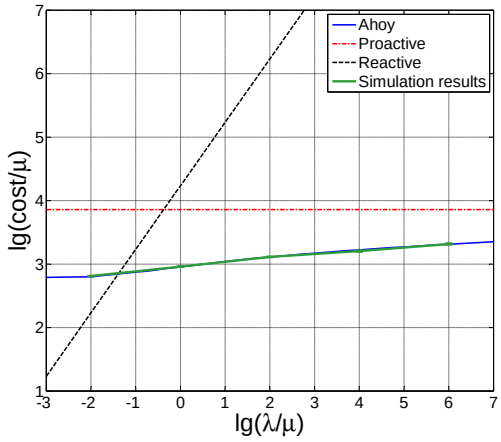


Figure 4.13: Overhead cost of `Ahoy`, the proactive and the reactive protocols for different λ/μ.

4.5.5 Summary

In this section, we observed the overhead network cost for analytical and simulation models in static ad-hoc networks. The overhead cost include both the periodic advertisements and false positive queries. In the analytical model, the traffic generated by false positive queries is estimated by using formulas of false positive probability. In the simulation, the number of false positive queries are counted in a straightforward way. The results from the analytical models are always within or very close to the 90% confidence interval of the simulation. We therefore conclude that both models give similar results. The approximate formula also provides very good estimates. The simple mathematical computation enables the approximate analytical model to be a good tool for computing the proper size of the attenuated Bloom filter

that yield minimal network costs. When we compare between `Ahoy` and traditional approaches, the same conclusion as in Section 4.4.3 can be drawn. The performance of `Ahoy` highly depends on the ratio between query and advertisement rates, but for a wide range of realistic parameter settings, it generates a significantly lower (up to an order of magnitude) traffic load than the proactive and reactive protocols.

Chapter 5

Dynamic Connectivity in Mobile Environment

In Chapter 4, we evaluated the performance of `Ahoy` in networks with static connectivity and discovered that `Ahoy` can save up to an order of magnitude of network traffic in most practical situations. It is therefore an effective solution for ad-hoc networks which are composed mostly of battery-supplied wireless devices without strong processing power.

However, ad-hoc networks are in general not static but dynamic in connectivity. This generates extra network load which may challenge the performance of discovery algorithms. In general, three causes of network dynamics can be identified: (i) nodes may be mobile; (ii) battery-supplied devices might exhaust their energy; (iii) the quality of the wireless transmissions might be varying due to unstable propagation conditions.

When a node is moving across the network, its set of neighbors is changing continuously. A temporary lack of energy supply can cause nodes to disappear, and worsening propagation conditions may result in broken links between nodes or lost packets. Similarly, due to changing conditions, nodes, and/or links can reappear.

If a link between two nodes disappears, those two nodes no longer consider each other as neighbors. If lost packets are keep-alive messages that inform about the presence of the node, other nodes in the network may consider this node to have been disappeared. If lost packets are updates, nodes continue routing packets with outdated information and generate redundant traffic. When links or nodes (re-)appear in the network, corresponding nodes need to add new neighbors.

In this chapter, we study how these dynamics influence the amount of extra traffic in the network. We examine both grid- and circular structured networks. We categorize the analysis based on the different categories of network dynamics that were mentioned above: link appearance and disappearance, node appearance and disappearance, packet loss, and node movement. We estimate the number of updates generated in `Ahoy` triggered by those dynamic factors and compare it with corresponding situations in the proactive and reactive protocols.

In Section 5.1, we discuss the probability to update ABFs when the availability of context information types is changed. Compared with circular-structured networks, grid networks have simple and predictable topologies. In Section 5.2, we therefore used grid networks to obtain insight in the extra amount of generated traffic due to dynamics. In Section 5.3, we then extend the analytical study from Section 5.2 to circular structured networks and validate the results with simulations. Finally, we compare `Ahoy` with the pro- and reactive protocols in Section 5.4. The work presented in Section 5.3 has been published in [53, 54].

5.1 Probability of Updating

In ABFs, different context information may share the same bit positions in the filter. As discussed in Chapter 4, this leads to false positive answers to queries. False positives may also occur when nodes update their ABFs. If a context information type is added or removed from a filter, there is a chance that the corresponding bits are also being used for other context information types. In that case, the ABF is not updated, even though the availability of context information types has actually been changed.

The probability of false positives has been derived in Section 4.2.3. A good approximation for the layer i false positive probability, $P_{fp,i}$, due to the addition of one context information type, has been shown in (4.30), as:

$$P_{fp,i} \approx (1 - e^{-b \cdot x_i / w})^b,$$

where x_i is the number of context information types in layer i before addition, b is the number of hash functions, and w is the width of the ABF. The probability that

there is no update when one context information type is added to layer i thus equals $P_{fp,i}$. We can also calculate the probability that there is no update when several context information types are added or removed. It is important to stress again that we assume that the hash functions are perfectly random. This implies that the false positive probabilities for two arbitrary (but different) context information types are independent of each other. If v_i is the number of added or removed context information types in layer i, the probability that there is no update in layer i, $P_{noupdate,i}$, is therefore equal to

$$P_{noupdate,i} = (P_{fp,i})^{v_i} . \tag{5.1}$$

As a consequence, it is thus possible, but very unlikely, that exactly the same bits are set for two sets of context information types, even if both sets contain many different context information types. If the modified context information types from layer k to layer l are independent of each other, the probability that no update occurs when context information types are modified in those layers, $P_{noupdate,e,f}$, equals the product of probabilities per layer, as:

$$P_{noupdate,k,l} = \prod_{i=k}^{l} (P_{fp,i})^{v_i} . \tag{5.2}$$

The probability of updating when context information types are added or removed, P_{update}, is equal to 1 minus the probability of no update.

$$P_{update} = 1 - P_{noupdate}. \tag{5.3}$$

Particularly, if all context information types from one node (in total, the number of types per node, s) have been added or removed from layer i, the update probability in this case equals:

$$P_{update}(i) = 1 - (P_{fp,i})^{s} . \tag{5.4}$$

If all context information types from one node have been added or removed from layer i to layer j of a node, the update probability of the node is:

$$P_{update}(i,j) = 1 - \prod_{k=i}^{j} (P_{fp,k})^{s} . \tag{5.5}$$

5.2 Grid Structure

We start our analysis with the simple grid structure, where every node has exactly 4 neighbors (see Section 4.1.1). We study the number of updates generated due to various dynamic connectivity scenarios: the disappearance and appearance of one link and one node, and mobility of one node, in the subsections that follow.

5.2.1 Link Disappearance

Due to interference or other environmental influences, the signal strength of a connection between two nodes may vary. If the signal strength is too weak, the connection is broken. When the signal strength improves, the connection is restored. In `Ahoy`, the following happens after a link is broken. The nodes that were connected by the broken link do not recognize each other as neighbors any more if they have not received a message for two consecutive keep-alive periods. As a result, both nodes follow **case 5** defined in Section 3.5 and update their filters and broadcast them to the other neighbors. The other neighbors within the query range propagate these updates.

The grid structure is a fully symmetric network structure. There are multiple paths between any pair of nodes. If a link has disappeared, there will always be another path via which two nodes can communicate. In most of the cases, the path length (i.e., the number of hops) will remain the same. For example, Figure 5.1 shows a grid-structured network with 9 nodes. Node A can reach Node D via Node B or Node C within two hops. In case the link between C and D is broken, A still can reach D via B within the same number of hops. Similarly, A can reach I within four hops via six different routes: $A \to B \to E \to F \to I$; $A \to B \to D \to F \to I$; $A \to B \to D \to H \to I$; $A \to C \to G \to H \to I$; $A \to C \to D \to H \to I$; $A \to C \to D \to F \to I$. If the link between D and F is broken, there are still four other routes between A and I. Since the structure of a grid-structured network is the same everywhere, this applies to every other node in the network. In Figure 5.1, A does not need to update the existence of D or I when link CD or DF is broken, because the information in D and I can still reach A with the same number of hops.

Only when a pair of nodes is located at the same row or column, there exists only one shortest path between them. For example, Figure 5.1 shows that if the

link between A and B is broken, A can not reach B within one single hop any more. A has to take the route via C and D to reach B after three hops. In this case, A removes the context information of B from layer 1 of its filter. Meanwhile, A can not reach E within two hops. Instead, the shortest path between A and E is via C, D, and F. The information of E is therefore removed from layer 2 of A. Similarly, if the link between B and E is broken, A can not reach E within two hops. A has to take a detour via C, D, and F to reach E. Again, A has to remove the context information of E from layer 2 of its filter and broadcast the updates.

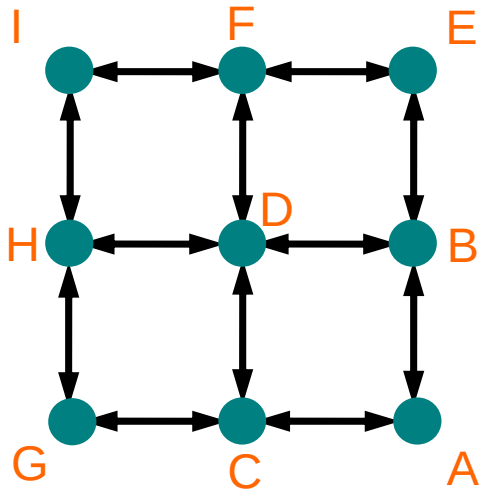


Figure 5.1: Connection in a grid-structured network.

Figure 5.2 demonstrates another example where the link between node A and B is broken. If the ABF has $d = 4$, 6 nodes in total need to update their filters. Updates due to the fact that A cannot reach B are represented by solid lines. Updates due to the fact that B can not reach A, are shown by dashed lines. Different colors denote the different number of hops: black for the first hop, orange for the second hop, and green for the third hop. A broken link results in updates for nodes that are in the same row or column, and that are within the communication range (within d hops) of the link. In Figure 5.2, when A detects that the link with neighbor B fails, A removes B's ABF from its cache and aggregates the remaining direct neighbors' (node I, F, K's) ABFs. By doing so, A removes B's information from its layer 1, C's information from its layer 2, and D's information from its layer 3, because nodes B, C, and D cannot be reached within the same number of hops anymore. If we

assume the context information types offered by node B, C, D are independent of each other and each node has s different context information types, based on (5.5), we can derive the update probability of node A as:

$$P_{update,A} = P_{update}(1,3) = 1 - \prod_{i=1}^{3} (P_{fp,i})^s \,.$$

Nodes F and G, which are on the same side of the disappearing link as A, are within $(d-2)$ hops away from A, and also need to update their filters (when $d = 4$). If we assume each node has s different context information types and the context information types from every node are independent of that from other nodes, for a node that is within i hops from node A, based on 5.5, the update probability equals:

$$P_{update}(i+1, d-1) = 1 - \prod_{j=i+1}^{d-1} (P_{fp,j})^s \,. \tag{5.6}$$

The same rule applies to node B and its related neighbors C and D. In case there would be no false positives, i.e., $P^s_{fp,j} = 0$, the number of updates is maximal, and equals to $2 \cdot (d-1)$. With the update probability for related layers, the expected number of updates when one link disappears, $N^g_{update\_ld}$, is:

$$E\left[N^g_{update\_ld}\right] = 2 \cdot \sum_{i=1}^{d-1} P_{update}(i+1, d-1) = 2 \cdot \sum_{i=1}^{d-1} \left(1 - \prod_{j=i}^{d-1} P^s_{fp,j}\right). \tag{5.7}$$

5.2.2 Link Appearance

When two nodes recover a lost link, they consider each other as a new neighbor and therefore establish the link from scratch. One of the two nodes receives a periodic keep-alive message from the other one, it follows **case 3** defined in Section 3.5, and replies with an update-request message. The update-request message triggers the new neighbor to broadcast its ABF. The node integrates the new filter and broadcasts its update. This update is forwarded to nodes within the advertisement range (d hops). For instance, the link between node A and B disappeared. After a while, the link quality improves and A receives a keep-alive message from B. A does not know B and sends an update request. B adds A as a new neighbor and

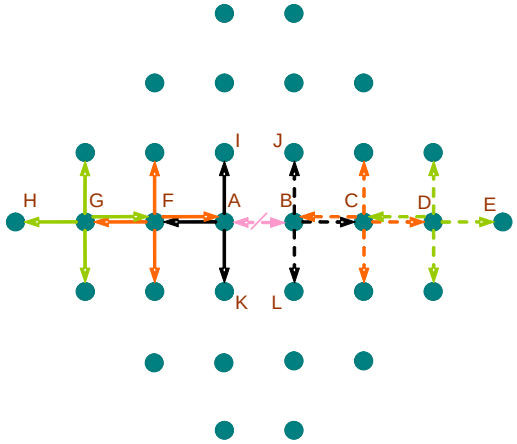


Figure 5.2: Link disappearance: the link between A and B disappeared with $d = 4$.

replies to A by sending its ABF. A receives the filter and adds B as a new neighbor. A integrates B's filter into its outgoing filter and broadcasts the update, which is propagated to nodes d hops away. Figure 5.3 demonstrates the sequence diagram of the example.

The new link generates paths that are shorter than the existing paths. These shortest paths are between nodes in the same row or column of the new link, but on opposite sides. The new link also generates paths between other pairs of nodes, but these are not shorter than the existing paths because of the highly symmetric structure of grid networks. As a result, only nodes along the same row or column need to update their filters. These nodes lie within $(d - 2)$ hops from the two end nodes of the new link. The update probability of a node can be achieved in the same way as in the case of link disappearance. If we assume that each node contains s different context information types which are independent of each other, the update probability is 1 minus the product of the false positive probabilities in the corresponding layers, as is described by (5.6). The expected number of updating packets due to the link appearance, $N^g_{update\_la}$, is the sum of update probabilities over the respective $2 \cdot (d - 1)$ nodes, plus one extra broadcast in response to the

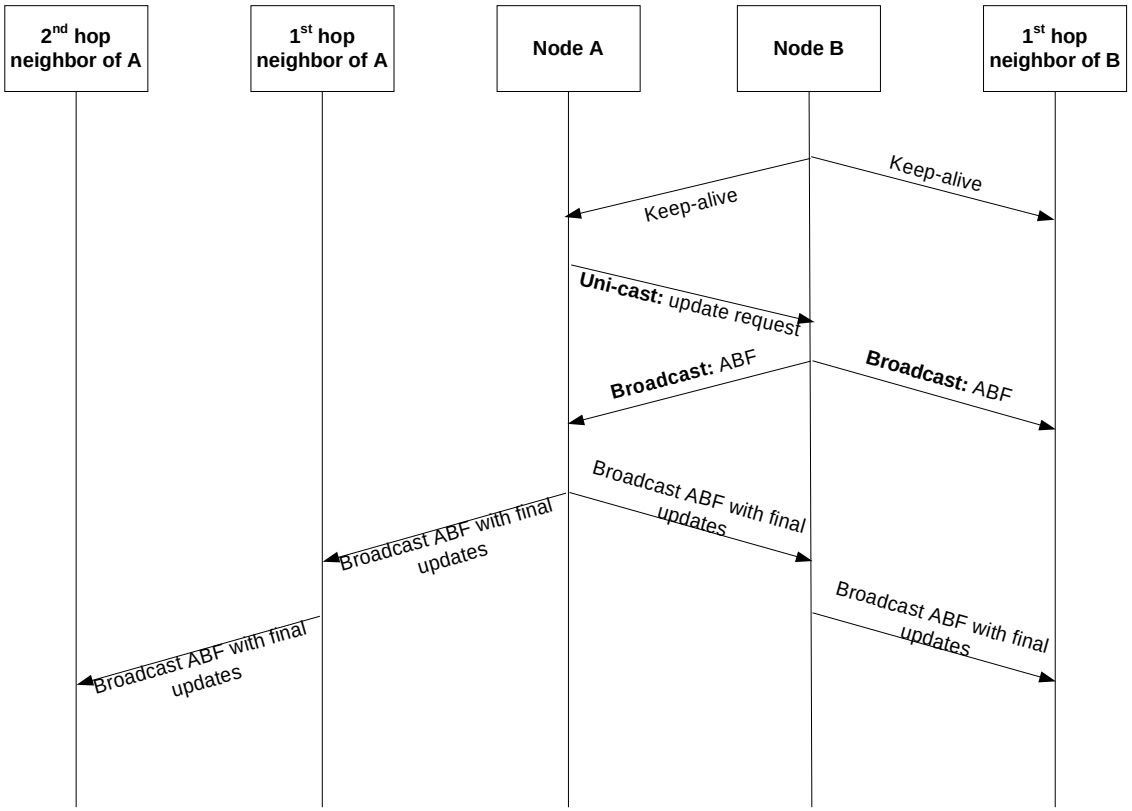


Figure 5.3: Link appearance: the link between A and B appeared.

update-request:

$$E\left[N^{g}_{update\_la}\right] = 1 + 2 \cdot \sum_{i=1}^{d-1} P_{update}(i, d-1) = 1 + 2 \cdot \sum_{i=1}^{d-1} \left(1 - \prod_{j=i}^{d-1} P^{s}_{fp,j}\right). \quad (5.8)$$

For example, the link between node A and B reappears, as shown in Figure 5.4. If $d = 4$, six nodes, including A and B, need to update their ABFs unless all layers show false positives. The updates triggered from node A are shown by solid lines and the updates triggered from node B are shown by dashed lines. One of nodes A or B, broadcasts one extra ABF, depending on which one receives the other's keep-alive message first.

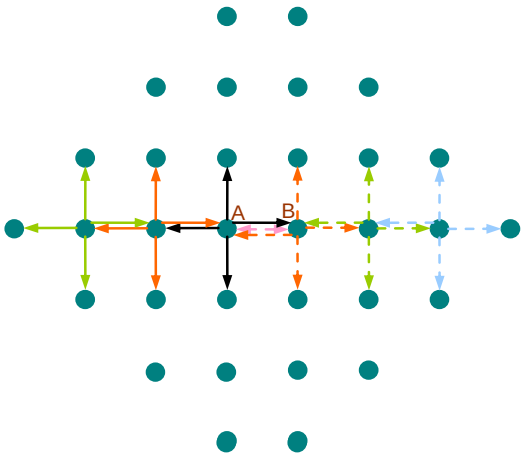


Figure 5.4: Link appearance.

5.2.3 Node Disappearance

A node may disappear from the network, due to various reasons, such as insufficient battery supply, non-functional antenna, system crash, and user actions like switching off the mobile or leaving the network.

In the current `Ahoy` protocol, the disappearance of a neighbor is noticed if no keep-alive message from this node has been received for two consecutive keep-alive periods. The neighbors of this absent node generate new ABFs, and broadcast them, based on **case 5** defined in Section 3.5. Due to the change, all nodes that receive the updated filter also regenerate a filter and broadcast it. This is repeated by nodes that are within $(d-1)$ hops of the disappearing node.

As we addressed in Section 5.2.1, there is always more than one path between two arbitrary nodes in a grid network. In particular, a node can reach another one in every two additional hops, starting from the minimum number of hops (shortest path). For instance, as is shown in Figure 5.1, one of the shortest path between node A and D is through B, which has a 2 hops distance. A can also reach D within 4 hops through the path $A \rightarrow B \rightarrow A \rightarrow B \rightarrow D$, and within 6 hops through the path $A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow D$, and so on. Certainly, those detours are not the most desirable and practical ones. However, this implies that the context information of node D appears in the ABF of node A, not only in layer 2, but also

in layer 4, 6, etc. Therefore, if node D leaves the network, node A needs to clean up the information from node D in all these layers. The information of the disappearing node is thus registered in layer 1 of its neighbors, in layer 2 of nodes two hop away, in layer 3 of both its direct neighbors and neighbors three hops away, in layer 4 of both neighbors two and four hops away, and so on. When node D disappears, its information has to be removed from the corresponding layers in all reachable nodes. In Section 4.1.2, we defined D_i^g as being the i-hop node degree in (4.1) and (4.2), as:

$$D_i^g = \begin{cases} 1, & \text{if } i = 0, \\ 4 \cdot i, & \text{if } i > 0. \end{cases}$$

In total, $\sum_{j=0}^{\lfloor (i-1)/2 \rfloor} D_{i-2j}^g$ number of nodes need to update their layer i. Each time the information is removed from one layer of a node, it causes one update with the update probability derived from (5.4). The expected number of updates for a given query range d, weighted by the update probability for the corresponding layers can be represented as follows:

$$\begin{aligned} E\left[N_{update\_nd}^g(d)\right] &= \sum_{i=1}^{d-1} P_{update}(i) \sum_{j=0}^{\lfloor (i-1)/2 \rfloor} D_{i-2j}^g \\ &= \sum_{i=1}^{d-1} (1 - P_{fp,i}^s) \sum_{j=0}^{\lfloor (i-1)/2 \rfloor} D_{i-2j}^g. \end{aligned} \tag{5.9}$$

5.2.4 Node Appearance

In this section, we observe the number of updates generated when a new node appears in an established grid-structured network. Suppose that one node appears in the square formed by nodes A, B, C, and D, as is shown in Figure 5.5. We refer to the appearing node as the new node. One basic assumption in this thesis is that the size of ABFs and hash functions are known by the new nodes. The method to distribute such information is beyond the scope of this thesis.

When a new node joins an existing network, the steps that were defined in Section 3.3.2 follow. It first broadcasts an ABF with only local information. The existing nodes receive the ABF and notice the new neighbor. Those nodes update their filters based on **case 4** defined in Section 3.5. Updates are propagated till d hops away from the new node. Meanwhile, the new node waits for a short moment

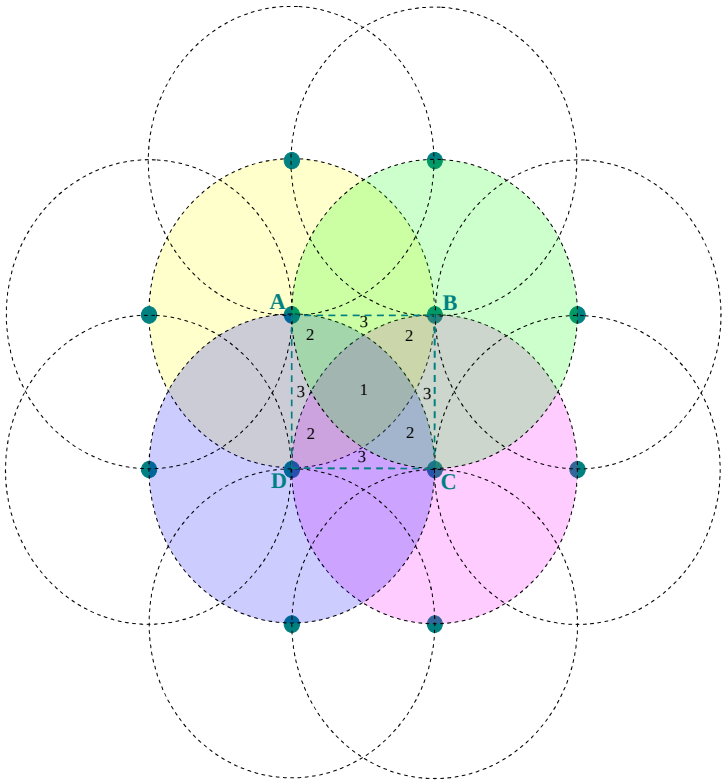


Figure 5.5: Node appears in a grid networks.

until it receives all the neighbors' replies. It aggregates all incoming filters, updates its own filter, and broadcasts it. This update does not need to be propagated further by neighboring nodes, because it does not consist new information. This is due to the fact that local information is duplicated in the lower layers of the first broadcasted ABF of the new node. Thus, every existing node only updates once to add information from the new node to the corresponding layers of its filter. The new node broadcasts its ABF twice, i.e., one original and one aggregated ABF. The sequence diagram of the updates is depicted in Figure 5.6.

The amount of network traffic depends on the number of neighbors of the new node. This number is directly related to the location of the new node. In a grid structure, we can divide the space of one grid into 3 different areas based on the number of direct neighbors the new node has, as is shown in Figure 5.5. We assume

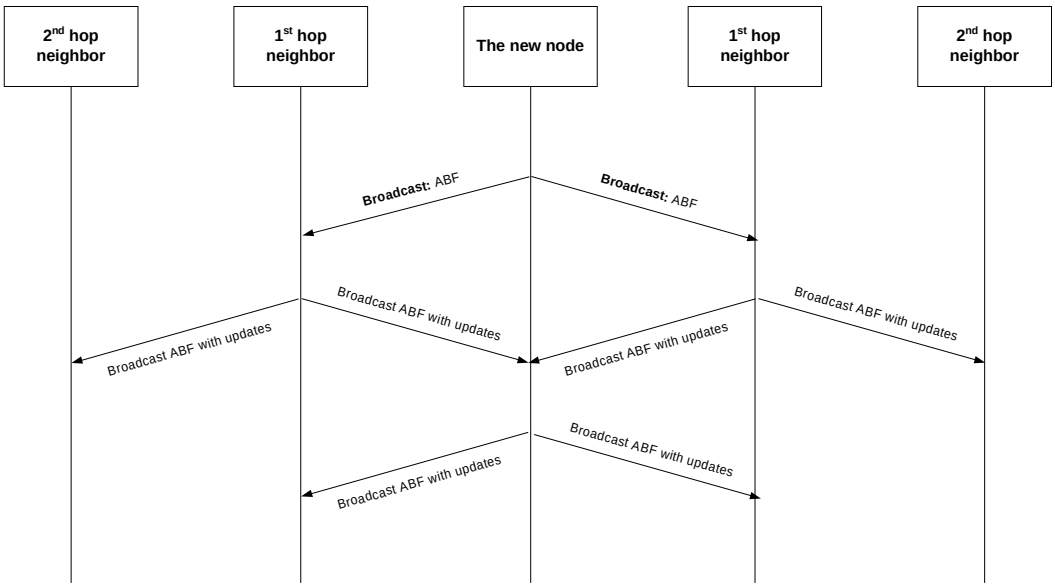


Figure 5.6: Sequence diagram of node appearance.

that the transmission range is equal to the length of one grid. The transmission ranges are illustrated by dashed circles. Nodes located within Region 1, can reach all 4 surrounding nodes A, B, C, and D. Nodes located in Region 2 can reach 3 nodes $\{A, B, D\}$, $\{A, B, C\}$, $\{B, C, D\}$, or $\{A, C, D\}$. Nodes located in Region 3 can reach only 2 nodes $\{A, B\}$, $\{B, C\}$, $\{C, D\}$, or $\{A, D\}$.

The expected amount of extra network traffic due to the new node can be obtained as the sum of the weighted number of updates $N^g_{update\_na,i}$ that occur when the new node is appearing in area i ($i \in \{1, 2, 3\}$). The weights are the probabilities, p_i, that the node appears in the corresponding areas. Thus, we have:

$$E\left[N^g_{update\_na}\right] = \sum_{i=1}^{3} N^g_{update\_na,i} \times p_i. \tag{5.10}$$

The number of updates per region i, $N^g_{update\_na,i}$, includes one ABF from the new node, which is the response to the update requests. In addition, all the existing

nodes within $(d-1)$ hops of the new node need to update their filters once. The ith hop neighbors of the new node insert the information into layer i to $(d-1)$ at once. Because of the context duplication, the bit positions which are set to 1 in the upper layers should also be set to 1 in the lower layers. If there is a false positive in an upper layer, this false positive also occurs in all lower layers. The false positive probability of adding information into layer i to $(d-1)$ is thus equal to the false positive probability of adding information into layer i. The update probability then equals the update probability of adding information of one node into layer i, as is described by (5.4).

The new node also needs to have an extra update to insert the information from its neighbors into its filter. It is almost certain that there will be no false positive in this case, because many information types (from all nodes within d hops) are added at the same time. We therefore consider that the new node needs to have an extra update with an update probability of 1.

In Region 1 in Figure 5.5, the new node has 4 neighbors, like any other node in the grid. The number of updates can thus be obtained in a similar way as in (5.9). In Region 2, the new node has 3 neighbors, and in Region 3, the new node has only 2 neighbors. Hence, the number of nodes that can be reached within i hops is not equal to D_i^g, but to $D_i^g - 1$ and $D_i^g - 2$ respectively. With the i-hop node degree D_i^g and s number of context information types per node, we can thus obtain the expected number of updates as:

$$E\left[N^g_{update\_na,1}\right] = 1 + 1 + \sum_{i=1}^{d-1} D_i^g (1 - P^s_{fp,i}). \tag{5.11}$$

$$E\left[N^g_{update\_na,2}\right] = 1 + 1 + \sum_{i=1}^{d-1} (D_i^g - 1)(1 - P^s_{fp,i}). \tag{5.12}$$

$$E\left[N^g_{update\_na,3}\right] = 1 + 1 + \sum_{i=1}^{d-1} (D_i^g - 2)(1 - P^s_{fp,i}). \tag{5.13}$$

We assume that the location of the new node has an uniform spatial probability distribution. Therefore, the probability that the new node appears in a specific region is equal to that region's area divided by the total area of one grid. If we assume that the length of the edge of a grid equals 1, the area of Grid ABCD,

$S_{ABCD} = 1$. Thus, as is illustrated in Figure 5.7, Region 1 is Region EFGH in Figure 5.7; Region 2 includes Region AFE, Region BGF, Region CHG, and Region DEH; Region 3 is composed of Region ABF, Region BCG, Region DHC, and Region AED.

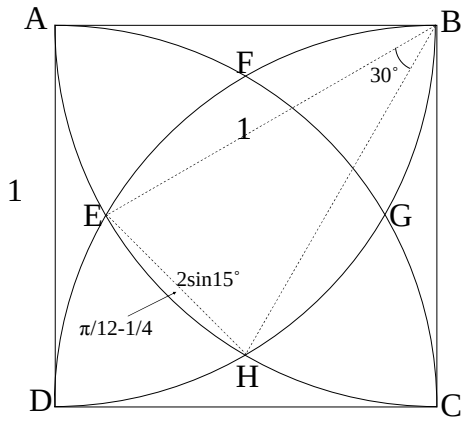


Figure 5.7: Grid ABCD.

According to basic trigonometry, we obtain the area of Region 1, 2, and 3, S_{A1}, S_{A2}, and S_{A3} as:

$$S_{A1} = (\frac{\pi}{12} - \frac{1}{4}) \times 4 + (2 \times \sin\frac{\pi}{12})^2 \approx 0.3151; \tag{5.14}$$

$$S_{A2} = \left((\frac{\pi}{4} - \frac{1}{2}) \times 2 - S_{A1}\right) \times 2 = \frac{\pi}{3} - 8\sin^2\frac{\pi}{12} \approx 0.5113; \tag{5.15}$$

$$S_{A3} = 1 - S_{A1} - S_{A2} = 2 - \frac{2\pi}{3} + 4\sin^2\frac{\pi}{12} \approx 0.1736. \tag{5.16}$$

The probabilities, p_1, p_2, and p_3, that the new node appears in the corresponding regions, are:

$$p_1 = S_{A1}/S_{ABCD} = ((\frac{\pi}{12} - \frac{1}{4}) \times 4 + (2 \times \sin\frac{\pi}{12})^2)/1 \approx 0.3151; \tag{5.17}$$

$$p_2 = S_{A2}/S_{ABCD} = (\frac{\pi}{3} - 8\sin^2\frac{\pi}{12})/1 \approx 0.5113; \tag{5.18}$$

$$p_3 = S_{A3}/S_{ABCD} = (2 - \frac{2\pi}{3} + 4\sin^2\frac{\pi}{12})/1 \approx 0.1736. \tag{5.19}$$

5.2.5 One Moving Node

The grid as a network model is a fixed structure. It does not allow nodes to move randomly through the network. However, we can consider a scenario that one additional node moves in the grid. The extra network traffic generated by the moving node depends on the path and the speed of the mobile node. We start the analysis with a simple scenario where the path of the moving node is a straight line, especially, it crosses an existing node A in the network. Due to the symmetric structure of a grid network, the paths yield identical results, when the new node moves from the same position as an existing node A under an angle of α, $\pi/2-\alpha$, $\pi/2+\alpha$, $\pi-\alpha$, $\pi+\alpha$, $3\pi/2-\alpha$, $3\pi/2+\alpha$, or $2\pi-\alpha$ with the horizontal, with $0 \leq \alpha \leq \pi/4$. This is illustrated in Figure 5.8. In this section, we first observe two extreme traces, with $\alpha = 0°$ and $\alpha = 45°$. When $\alpha = 0°$, the node crosses the network along the grids. When $\alpha = 45°$, the node crosses the network diagonally. We finish this section with traces in arbitrary directions. To simplify the problem, we assume that the node moves slowly enough to finish the updating process with the neighboring nodes.

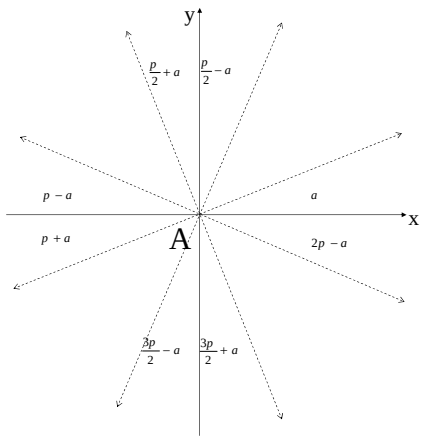


Figure 5.8: The mobile node move across node A.

Transverse angle: $\alpha = 0°$

In this scenario, a node traverses horizontally through the network. We assume that the distance between direct neighbors is exactly equal to the transmission range of the nodes, which is illustrated in Figure 5.9. Suppose that a mobile node moves along the horizontal edges of the grids. Most of the time, it is in the transmission range of two nodes. For example, the mobile node M is in the transmission range of node D and E, until it passes node E. After passing E, it is out of range of D, but within the range of C, as is shown in Figure 5.9. A new direct connection with C is established. Meanwhile, the direct connection with node D is lost. At the moment that the mobile node M is at the same location as E, M is in the transmission ranges of 5 nodes, including A, B, C, D, and E. This location is the only location where the transmission ranges (shown by circles) of four nodes, i.e., A, B, C, and D, intersect. Because the mobile node moves, the time spent at location E is infinitely small. The probability that the mobile node receives a packet from the nodes A and B within this infinitely small time period is infinitely close to zero, vise versa. Therefore, we consider that the mobile node and the nodes A and B do not recognize each other as a new neighbor within this time period.

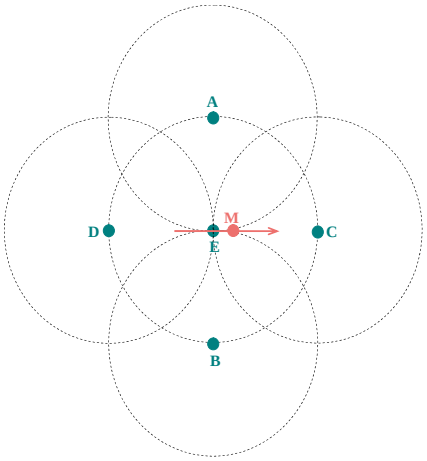


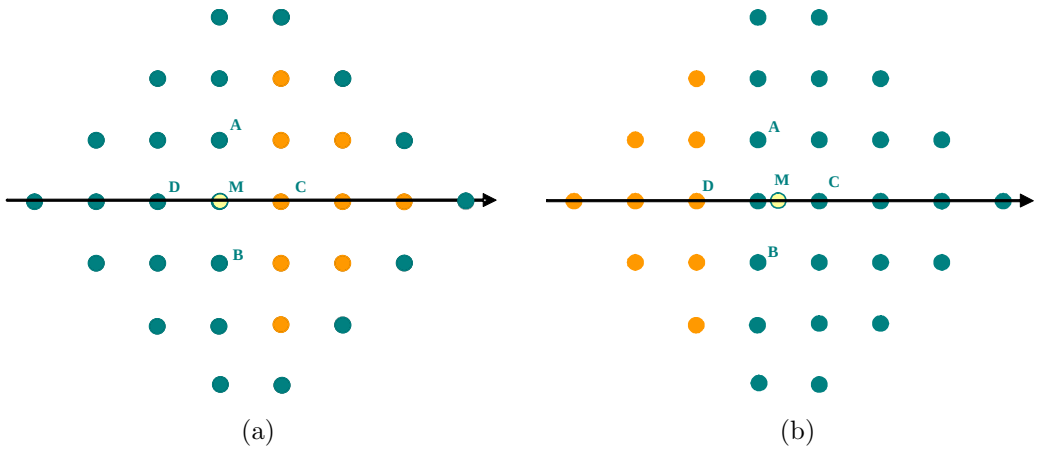
Figure 5.9: Mobile node moving along the edges of a grid.

The new connection with C generates a new shortest path between the mobile node M and node C. It also generates new shortest paths between node M and all

the nodes d hops away that are in the same "column" of C or in the columns on the right side of C. To add the related information for those new shortest paths, M and nodes $(d-1)$ hops away from M need to update their filters, based on **case 3** defined in Section 3.5. These nodes form an equilateral triangle with C in the middle of the long side, and with the right angle at the node $(d-1)$ hops away from M in the direction in which M is moving. Similarly, new shortest paths are established on the other side, because M lost the connection with D. These nodes are in a similar triangle, in which D is in the middle of the long side, and in which the right angle points to the opposite direction to which M is moving. For $d = 4$, in Figure 5.10 the orange nodes represent the nodes that need to be updated due to the new connection with C (Figure 5.10(a)) and due to the lost connection with D (Figure 5.10(b)). Those nodes need to update once to remove information of M in their corresponding layer, based on **case 5** defined in Section 3.5.

As a consequence, the mobile node M updates its filter by changing the new shortest path information into the related layers. During the update, M aggregates the ABFs from its new direct neighbors. In doing so, it also modifies the shortest paths to other nodes, which pass through the new neighbors and the lost neighbors. The update of M can therefore be done at once. Note that again we give a mild assumption that the update probability is equal to 1, because many context information types (from several nodes) are added or removed at once. Simultaneously, any node which can reach M in i hops via a new connection or cannot reach M anymore in i hops due to a broken connection also need to respectively add information to or remove information from its layer i. According to Figure 5.10, in which $d = 4$, there are two times $(2i-1)$ number of nodes in the ith hop. The number of updating nodes in the ith hop is also equal to $D_i^g - 2$, because for a grid structure $D_i^g = 4i$. The expected number of updates, weighted by the update probability (5.4), plus one reply to the update request, can be obtained as:

$$E\left[N^g_{update\_hori}\right] = 1 + 1 + \sum_{i=1}^{d-1}(D_i^g - 2)(1 - P^s_{fp,i}). \tag{5.20}$$



(a) (b)

Figure 5.10: Horizontal move: (a) the case when new shortest paths are established. (b) the case when the shortest paths are eliminated. In both figures, orange nodes are the ones that need to update their filters. The hollow node is the mobile node M.

Transverse angle: $\alpha = 45°$

In this case, the trace of the mobile node is exactly along the diagonal of a grid. For example, as is shown in Figure 5.11, the mobile node is moving along the diagonal AC of Grid ABCD.

Based on the different sets of direct neighbors, the trace in one grid can be divided into 3 sections:

- Section 1: The mobile node can not reach node C directly. The set of its direct neighbors is $\{A, B, D\}$.
- Section 2: The mobile node can reach all 4 nodes directly. The set of its direct neighbors is $\{A, B, C, D\}$.
- Section 3: The mobile node can not reach node A any more. The set of its direct neighbors is $\{B, C, D\}$.

Updates are triggered by changes in ABFs, which occur when a node is added or removed from the original ABFs, based on **case 3 and 5** defined in Section 3.5. In this scenario, the changes occur when the mobile node M (the purple node in

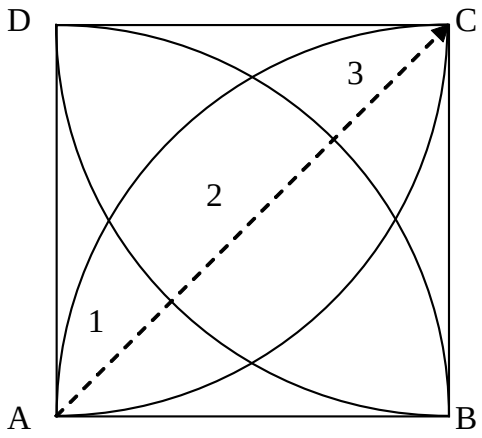


Figure 5.11: 45° across the grid (diagonal).

Figure 5.12) travels from one section to another. When the mobile node M moves from Section 1 to 2, node C is the new neighbor of the mobile node. As a result, new shortest paths are generated from the mobile node and node C, and nodes beyond C. Those nodes need to update their own filters due to the new shortest path to the mobile node. If we set the depth of the filter d at 4, the orange nodes in Figure 5.12 are the nodes which need to update their Bloom filters.

Similarly, if the mobile node moves from Section 2 to 3, it loses the direct connection with node A. As a result, shortest paths between the mobile node and node A, and nodes beyond A are changed. Since nodes only remove the information related to the shortest path, every related node needs to update only once. As is shown in Figure 5.13, the orange nodes are the nodes that need to update Bloom filters when d equals 4.

From this analysis, we can deduce that updates only happen to the nodes of which shortest path to the mobile node have been changed. Due to the special network structure, the updating area is a right-angled triangle. The right angle is at the location of the node with which a new connection is established or with which a connection is broken. The right angle of this triangle is faced towards the grid in which the mobile node is moving. The number of nodes located in the catheti equals to $(d-1)$. In this thesis, we name this updating area an "update triangle".

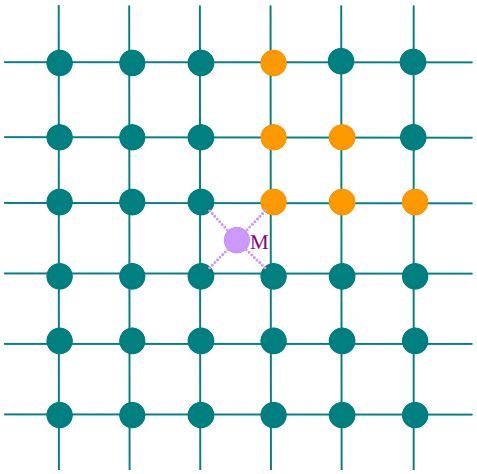


Figure 5.12: Updates for a node moving from area 1 to 2.

The nodes that are located in the "update triangle" need to update their filter once to modify the shortest path information. The number of i-hop neighbors in the "update triangle" is equal to i. The ith hop nodes update their layer i by adding (in case of a new connection) or removing (in case of a broken connection) information types of the mobile node. This is done with the update probability, described by (5.4). Again, we assume the the update probability is 1 for the update of the mobile node. As explained earlier, all nodes, including the mobile node, only need to update once. As we discussed in Section 5.2.2, the updates are always triggered by an update request when a new connection is established. This generates one extra broadcast. For a broken connection, no extra traffic is needed to trigger the updates. The expected number of updates for a diagonally trace across a grid, $N^g_{update\_diag}$ can thus be achieved as:

$$\begin{aligned} E\left[N^g_{update\_diag}\right] &= 2 \cdot \left(1 + \sum_{i=1}^{d} i \cdot P_{update}(i)\right) + 1 \\ &= 2 \cdot \left(1 + \sum_{i=1}^{d} i \cdot (1 - P^s_{fp,i})\right) + 1. \end{aligned} \tag{5.21}$$

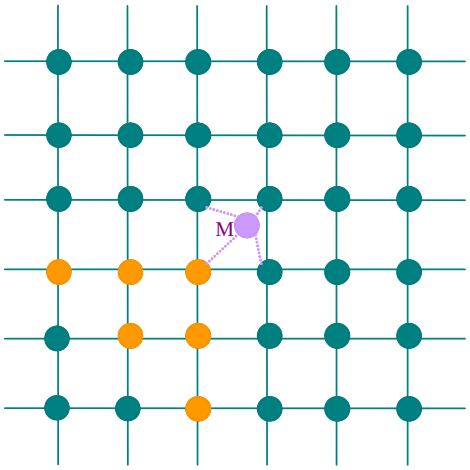


Figure 5.13: Updates for node moving from area 2 to 3.

Movement in arbitrary directions

So far, we have analyzed how many expected updates are needed when a node moves along edges or diagonally across a grid. Extra traffic is generated due to changes in connections between the mobile node and neighboring nodes. As a result, shortest paths between the mobile node and other nodes are generated or eliminated. Therefore, those nodes need to update their attenuated Bloom filters. Interestingly, we can define so-called "update triangles" that contain the nodes which have to update their filters. These triangles can be defined such that they have the following characteristics:

1. Given that d is the depth of the ABF, an "update triangle" is a right-angle triangle with $(d-1)$ number of nodes in both catheti.

2. Both catheti are parallel to grid edges (horizontal and vertical).

3. The right angle of the triangle is at the node with which a connection is established or broken.

4. The right angle of the triangle is faced towards the grid in which the mobile node is moving.

When a connection is established or broken, the mobile node and nodes in the "update triangle" need to update once. The nodes in the "update triangle" which are i hops away from the mobile node, update their layer i by respectively adding or removing the information types from the mobile node. This is done with update probability, described by (5.4). We assume that the update probability of the mobile node is 1 in this case, because it has to modify many context information types at once. No advertisement loops are created during updating. When the mobile node aggregates ABFs from its new neighbors in its own ABF, updates for all other nodes that can be reached via the new neighbors, are automatically done. When a connection is established, the changes are triggered by an update request. As a result, there is one extra broadcast. When a connection is broken, there is no such broadcast needed, because the updates are simply triggered by not receiving information (advertisements or keep-alive messages) from the neighbor anymore.

From the updating process, described above, we can now generalize the expected number of extra packets when a new connection is established, as:

$$E\left[N^{g}_{update\_add}\right] = 1 + \sum_{i=1}^{d} i \cdot P_{update}(i) + 1, \tag{5.22}$$

and the expected number of extra packets when a connection is broken, as:

$$E\left[N^{g}_{update\_remove}\right] = 1 + \sum_{i=1}^{d} i \cdot P_{update}(i). \tag{5.23}$$

The horizontal transverse along edges is an extreme case here. The corresponding update area consists of two "update triangles" that have one catheti in common, as is illustrated in Figure 5.10. The reason for this is that the mobile node moves on the edge of two grids, and hence generates two update triangles when it establishes a new connection or looses one. When the mobile node moves along the edge of the grids, there are always two "update triangles" that share one cathetus, which is aligned with the edge along which the node is moving.

The location of the mobile node in the grid determines which nodes are its direct neighbors. Based on this, we can divide a grid into 9 sections as is shown in Figure 5.14. For a grid formed by 4 nodes A, B, C, and D, Table 5.1 lists the direct neighbors of the mobile node given its location in the grid.

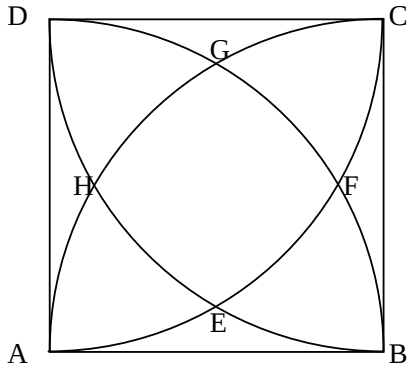


Figure 5.14: Grid division.

When the mobile node moves from one region to the other, connection(s) with neighboring node(s) are established or broken. The nodes in the corresponding update triangle(s) need to update their filters accordingly. This principle is always valid, no matter in which direction the mobile node is moving.

For example, in Figure 5.15 the mobile node moves along a trace crossing the following regions in succession: $R_{AHD} \rightarrow R_{AEH} \rightarrow R_{EFGH} \rightarrow R_{FCG} \rightarrow R_{DGC}$. The trace of the mobile node is represented by the dashed line. When the mobile node moves from R_{AHD} to R_{AEH} and from R_{FCG} to R_{DGC}, it establishes and breaks the connection with node B, respectively. This is shown by red solid arrows in Figure 5.15(a). The mobile node first needs to add and then remove the information type from node B from its filter. Meanwhile, nodes for which the shortest path to the mobile node have changed, need to update their filters as well. Those nodes are located in the "update triangle" of B, which right angle is located at node B.

When the mobile node moves from R_{AEH} to R_{EFGH}, it establishes a link with node C. This is illustrated in Figure 5.15(b). This triggers the mobile node to update its filters by adding information from the new neighbor. Meanwhile the nodes from the "update triangle" of C, have to update their filters as well (shown in blue in Figure 5.15(b)). When the node moves from R_{EFGH} to R_{FCG}, the direct connection with node A is broken. Node M and the nodes located in the "update triangle" of A (shown in red in Figure 5.15(b)), update their filters accordingly. Based on results

Table 5.1: Reachability of a mobile node that is located in specific regions of Grid ABCD).

Region	Direct Neighbors
R_{AEH}	A, B, D
R_{AHD}	A, D
R_{EFGH}	A, B, C, D
R_{HGD}	A, C
R_{FCG}	B, C, D
R_{DGC}	C, D
R_{AEB}	A, B
R_{EFB}	A, B, C
R_{CBF}	B, C

from Scenario "transverse moment: $\alpha = 45°$", we can derive the number of updates. In this case, there are in total two new connections that have been established (to B and C) and two that have been broken (to A and B). Based on (5.22) and (5.23), we thus can derive the total expected number of updates as:

$$\begin{aligned} E\left[N^g_{update\_example}\right] &= 2 \cdot N^g_{update\_add} + 2 \cdot N^g_{update\_remove} \\ &= 4 \cdot (1 + \sum_{i=1}^{d} i \cdot P_{update}(i,i)) + 2 \\ &= 4 \cdot (1 + \sum_{i=1}^{d} i \cdot (1 - P^s_{fp,i})) + 2. \end{aligned} \tag{5.24}$$

5.2.6 Summary

The analytical studies in grid-structured networks provide us an overview of how updates are triggered and propagated in various situations: disappearance and appearance of both links and nodes, as well as in case of a node moving through the grid. We can derive the following conclusions from our study in grid-structured networks:

- Updates happen whenever a shortest path has been established or has disappeared.

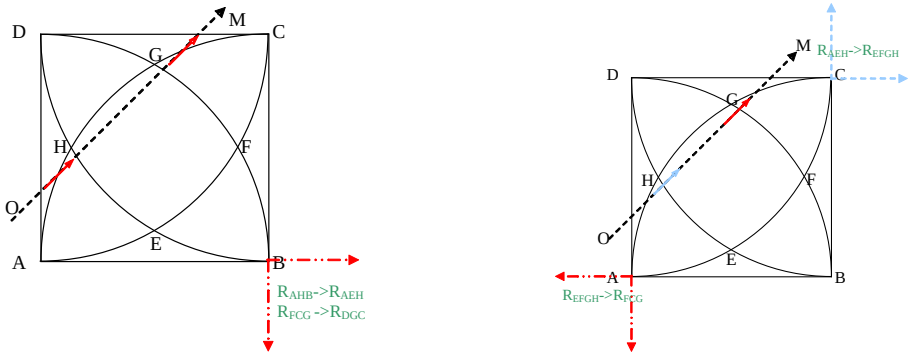


Figure 5.15: An example of a mobile node crossing a grid.

- When a link or a node disappears, corresponding nodes update their ABFs based on **case 5** defined in Section 3.5. When a link appears, **case 3** defined in that section has been followed. When a node appears, corresponding nodes update based on **case 4** defined in that section, and the new appearing node follows steps defined in Section 3.3.2.

- When a link appears or disappears, the existing shortest paths containing the changed link need to be updated. Due to the highly symmetric structure of grid networks, updating nodes are aligned to the changed link. Only one update is required for each node.

- When a node appears or disappears, nodes within d hops range of the appearing/disappearing nodes need to update accordingly. To remove a context information type in the network, nodes need to update their ABFs several times to remove information layer by layer. To add a context information type in the network, related nodes need to update only once, due to context duplication.

- When a node moves in the network, we discover that the updating nodes are located in an "update triangle", in which the node at the right angle is the node with which the mobile node establishes or looses a connection. No matter how nodes move in grid-structured networks, when a connection is established

or broken, the mobile node and the nodes in the related "update triangle" update only once to modify their ABFs accordingly.

5.3 Circular Structure

In this section, we extend our dynamic connectivity study to circular-structured networks. We investigate four different cases: a node disappearing, appearing, losing some packets, and traveling through the network. For the first three cases, we use an analytical approach to quantify the extra traffic load which is caused by updates, and we validate the results by simulations. For the mobility scenario, we analyze the case that one node travels in a straight line through the network. We do simulations for different speeds and different network densities. In particular instances, we could also derive the results analytically. For these cases, we compare the simulation results with the analytical ones.

We do not consider the scenario of link disappearance and appearance. In that scenario only pairs of nodes are updated, of which shortest paths pass through the modified link. For some topologies, one link could provide shortest paths between several nodes. In a grid network, for example, there exist unique shortest paths between pairs of nodes that are aligned with the modified link. Yet, the number of nodes affected is small compared to that in a scenario where a node is appearing, disappearing or moving through the network. Another example in which a link is quite important is when it connects two clusters of nodes. However, we do not consider such extreme topologies. We assume that nodes are randomly distributed. In that case, multiple shortest paths often exist between two nodes. When the density is high, only nodes that are directly connected by the modified link need to be updated. Therefore, in general link appearance or disappearance will only lead to a marginal increase in traffic.

5.3.1 Simulation Setup

The circular structured networks are modeled in the discrete event simulator OPNET Modeler version 11.5 [63]. Each node has a 300 meters communication range r. According to (4.4), if the study area is $1700m \times 1700m$, we need at least 61 nodes

to generate 1-connected graphs with 90% (confidence) probability. To include both non-connected and fully connected graphs, we observe the scenarios with 25, 61, 100, 125, 150 nodes, randomly distributed in the study area. The network densities, the number of nodes per m^2, are 0.87×10^{-5}, 2.11×10^{-5}, 3.46×10^{-5}, 4.33×10^{-5}, and 5.19×10^{-5}, respectively. We consider the 25 nodes scenario as a low density network, and the 150 nodes scenario as a high-density network. The node that disappears, appears, or is temporarily unreachable, is located in the center of the area to avoid border effects. For each parameter setting, 30 independent runs are performed for estimating the 90% confidence interval.

Some basic ABF parameters are set as follows: number of hash functions per service, $b = 10$; ABF width, $w = 1024$bits; ABF depth, $d = 3$; number of context sources advertised per node, $s = 1$.

5.3.2 Node Disappearance

We start the analysis with the case of one node disappearing from the network. Similarly as in grid-structured networks, when a link or a node disappears, corresponding nodes update their ABFs based on **case 5** defined in Section 3.5. We first observe the actions of one node when one of its direct neighbors disappears and then study the total number of updates due to the disappearance of that node. We assume that some node A disappears at the moment the network has reached a stable state, i.e., all ABFs are up-to-date at that moment. Node B is one of the neighbors of node A. When node A and B are direct neighbors, the communication range of nodes A and B intersect with each other, as is shown in Figure 5.16. Nodes that are located within the intersection area are direct neighbors of nodes A and B. All these nodes, including B, can reach A in one hop, but also in two hops, and probably in even more hops. Information of A is therefore not only in layer 1 of B, but also in its lower layers. To reduce extra traffic due to the advertisement loop, context information types are duplicated in the lower layers anyway.

At the moment that A disappears, direct neighbors, such as B, notice that the connection with A is broken when no keep-alive message has been received for two consecutive keep-alive periods. As a result, they remove the information from A from their layer 1. However, not all direct neighbors will act simultaneously, because updates are unsynchronized to avoid collisions. Suppose that B has updated its

information in layer 1. It may still think that it can reach the information of A through nodes in the intersection area that have not yet updated their layer 1. In fact, this is exactly what happens when only the link between A and B is broken (but A has not disappeared). Only when all the nodes in the intersection area have updated their layer 1, node B can update its layer 2.

Suppose now that B is the last node in the intersection area that updates its layer 1. At that time, B thinks that it can still reach A in at least three hops, but not within two hops. In that case, B can update its layer 1 and 2 at once. In general, some other nodes that are not in the intersection area of A and B may also update two layers at once. The number of such nodes depends on the way the updates propagate. If the update order is random, some nodes get isolated, i.e., they are updated after all their neighbors in the intersection area with A have been updated. If the updates propagate in one direction, for example clockwise, in the most extreme case, only the last node in the sequence gets isolated and updates two layers at the same time.

Similarly, after all 2-hop neighbors of A have removed the information of A from layer 1 and 2, nodes that could reach A in three hops, will realize that A is not available within three hops anymore, and they will clean layer 3. Again, at least one node will clean up two layers, i.e., layer 3 and 4, at once. This process is continued until nodes have cleaned up the information from A in all corresponding layers.

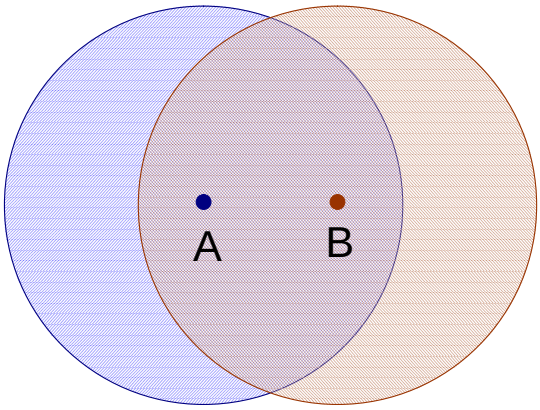


Figure 5.16: Node A and B are direct neighbor of each other.

From this, we can estimate the number of updates that are generated when one node disappears. Since a high density network which is $k-$connnected ($k > 1$) is assumed here, all ith hop neighbors of the disappearing node can also reach it in any number of hops which is larger than i. This implies that the total number of nodes that can reach the disappearing node in exactly i hops is $\sum_{j=1}^{i} D_j$. Nodes, which can reach the disappearing node in one hop, first clean up their layer 1. In total, there are D_1 of such nodes. Then all nodes, which can reach the disappearing node in 2 hops, clean up their layer 2. In total, there are $(D_1 + D_2)$ of such nodes. This process continues till layer $(d-1)$ after which all neighbors within at most $(d-1)$ hops have cleaned up their filters. However, as we have discussed in Section 4.1.2, we can only obtain an expected value of the i-hop node degree for circular-structured network, $E[D_i^c]$, as is shown in (4.13):

$$\lim_{n\to\infty} E[D_i^c] = \begin{cases} 1, & i = 0, \\ n\left(\pi(ir)^2 - \pi\left((i-1)r\right)^2\right) = (2i-1)n\pi r^2, & i > 0. \end{cases}$$

Further, except for the one hop neighbors, not all i hop neighbors ($2 \leq i \leq d-1$) need to update their layer i. As we addressed above, there is at least one node that has cleaned layer i simultaneously with layer $(i-1)$. Here, we achieve an upper bound, because we consider there is only one such node for each hop between 2 and $(d-1)$. Therefore, there are in total $(\sum_{j=1}^{i} E[D_j^c] - 1)$ number of i hop neighbors ($2 \leq i \leq d-1$), which may update layer i. Every i hop neighbor cleans layer i with an update probability of $(1-(P_{fp,i})^s)$. We can thus derive the expected number of clean-up updates for one node disappearing, $E[N^c_{updates\_nd}]$, to be smaller or equal to the upper bound:

$$E[N^c_{updates\_nd}] \leq E[D_1^c] \cdot \left(1-(P_{fp,1})^s\right) + \sum_{i=2}^{d-1} \left(\left(\sum_{j=1}^{i} E[D_j^c] - 1 \right) \cdot (1-(P_{fp,i})^s) \right). \tag{5.25}$$

We verify the results of this approximation with simulations. Table 5.2 presents the results for the 5 different network densities. In the table, we also show the relative error, which is defined as the relative difference between the analytical result V_a and

the mean of the simulations $\bar{V_s}$:

$$Error = \frac{V_a - \bar{V_s}}{\bar{V_s}} \times 100\%. \tag{5.26}$$

Table 5.2: Results on node disappearance in circular-structured networks.

Density ($\times 10^{-5}$ $\frac{node}{m^2}$)	Upper bound	Mean Simulation Results	Confidence Interval	Error (%)
0.87	11.23	5.63	(4.54, 6.72)	99.47
2.11	28.84	20.03	(18.08, 21.98)	43.98
3.46	47.92	39.40	(37.30, 41.50)	21.62
4.33	60.14	51.70	(48.27, 55.12)	16.32
5.19	72.36	66.37	(62.54, 70.20)	9.03

Table 5.2 shows that the relative error decreases with density. It drops from 99.47% to 9.03% when the network density increases from 0.87×10^{-5} node/m^2 to 5.19×10^{-5} node/m^2. This result suggests that the difference between upper bound and simulation is due to our assumption of a very high-density network. The estimation of the i-hop node degree from (4.13) is slightly overestimated by the analytical model. In the simulations, fewer nodes are involved in updating, and therefore less traffic is generated. Also, more nodes in the ith hop can clean up more than one layer at once, whenever all of its direct neighbors have already removed the related information from layer i of their filters. This results in fewer update packets than is estimated by the upper bound. However, the fraction of nodes that can update two layers at once is small for high densities. We therefore conclude that the upper bound is a relatively good estimate for high density networks.

In Figure 5.17, we further show the results. The analytical results are shown by the solid line, while the simulation results are shown by the dashed line. The error bars represent the confidence intervals. The figure nicely illustrates how much the upper bound overestimates the extra traffic that is generated when a node disappears. We also obtain the number of update packets for the grid-structured networks from (5.9). We estimate the network density in [52] as 0.7×10^{-5}. This yields 12.0 updates transmitted for each disappearing node, which is plotted as a

star in Figure 5.17. This number is slightly higher than the one for a circular network with the same density. This difference can be explained by the difference in i-hop node degree for both network structures.

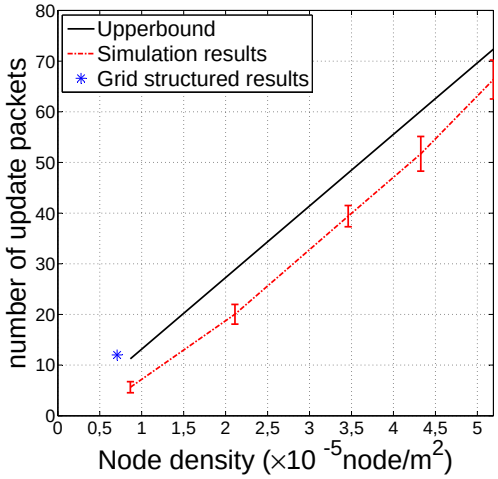


Figure 5.17: Number of updates generated when a node disappears.

5.3.3 Node Appearance

In this section, we consider the scenario that one new node appears in the network. Similar as in grid-structured networks, a new node follows the steps described in Section 3.3.2, and other corresponding nodes follow **case 4** defined in Section 3.5 for updates. The new node first broadcasts an ABF with only local information. The direct neighbors receive the broadcast and update their filters. All nodes till $(d-1)$ hops away from the new node update their ABFs accordingly. The new node waits for a short moment and aggregate ABFs from all its neighbors and generate a new updated outgoing ABF. The sequence diagram is shown by Figure 5.6.

We assume a high density network, which implies that the appearance of the new node does not generate shorter paths between existing nodes. Therefore, for any node up to $(d-1)$ hops away from the new node, only the appearance of the new node is added into the existing filters. Further, since the local information of the new node is duplicated to every layer of its ABF, the neighbors can update

their filters at once rather than adding the information layer by layer. After the initial broadcast (reply for the keep-alive message), every node, including the new one, therefore only updates once. Similar as in Section 5.2.4, the update probability equals to the update probability of adding s context information types into layer i (see (5.4)). The expected number of updates can thus be quantified as the total number of nodes within $(d-1)$ hops, weighted by the update probabilities, plus two broadcasts of the appearing node:

$$E[N^c_{updates\_na}] = \sum_{i=1}^{d-1} E[D^c_i] \cdot (1 - P^s_{fp,i}) + 1 + 1. \tag{5.27}$$

We compare the analytical result with the simulation. The results are shown in Figure 5.18 and Table 5.3. The figure and table demonstrate that the analytical and simulation results compare relatively well. Also in this case, the error between analytical and simulation results decreases with density, i.e., from 38.59% to 1.73% when the network density increases from 0.87×10^{-5} node/m^2 to 5.19×10^{-5} node/m^2. In high density scenarios like 4.33×10^{-5} $\frac{node}{m^2}$ and 5.19×10^{-5} $\frac{node}{m^2}$, the analytical results fall within the confidence interval of the simulations. There are two opposite effects that influence the accuracy of the analytical estimation. Both of them follow from our assumption of very high densities. First, the ith hop node degree is overestimated in the analytical model, especially for low densities. The analytical model therefore predicts too much traffic. However, the appearance of a new node will also not yield new shortest paths in the analytical model. In reality, new shortest paths will be generated, especially in low density environments. As a result, quite a few nodes need to update more than once, because new i-hop neighbors are discovered. Figure 5.18 shows that the latter effect does not completely compensate for the former effect. Of course, when the network density is high, the biases in the analytical model become less severe.

The corresponding results of the cost for the grid structure, as obtained from Formula 5.10, is plotted as a star in Figure 5.18. Again, the node density is 0.7×10^{-5}. The result is very comparable to the analytical result for the circular structured network.

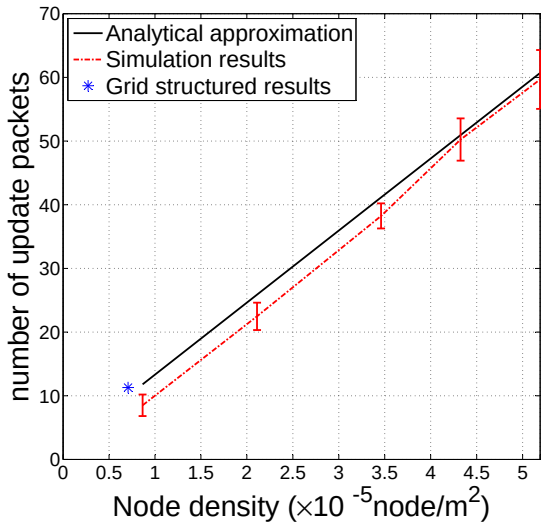


Figure 5.18: Number of updates generated when a node appears.

5.3.4 Packet Loss

In this section, we study the situation where some packets of a node get lost, e.g., due to unfavorable propagation conditions in all directions. Note that this is different from a broken link, because in this case, not one, but all neighbors cannot receive the packets anymore. If at least two consecutive keep-alive messages from a node are lost, the neighbors of the node consider the node to have disappeared. They start cleaning up their ABFs as we described in Section 5.3.2. After some time, the transmission quality of the node improves, and keep-alive messages are received again. The neighbors think that a new node has appeared in the network. The actions, described in **case 3** in Section 3.5, are taken. The sequence diagram is shown in Figure 5.19.

Here, we assume that the packet loss only occurs in one direction, i.e., we assume that the node can still receive packets from its neighbors. This is slightly different than the scenario in Section 5.3.3. In Section 5.3.3, the new node does not have any knowledge about the network, while in this case the (appearing) node keeps receiving updated information from the neighbors. Therefore, the appearing node does not need to update its filter to insert information types from its neighbors.

Table 5.3: Results for node appearance in circular-structured networks.

Density ($\times 10^{-5}$ $\frac{node}{m^2}$)	Analytical Results	Mean Simulation Results	Confidence Interval	Error (%)
0.87	11.78	8.50	(6.81, 10.20)	38.59
2.11	25.87	22.47	(20.31, 24.62)	15.13
3.46	41.13	38.23	(36.26, 40.20)	7.59
4.33	50.92	50.23	(46.91, 53.56)	1.37
5.19	60.70	59.67	(55.05, 64.29)	1.73

One update less is thus generated than in Section 5.3.3. The number of updates generated in this scenario, $N^c_{packet\_loss}$, can be obtained by:

$$N^c_{packet\_loss} = N^c_{updates\_nd} + N^c_{updates\_na} - 1. \tag{5.28}$$

From (5.25), (5.27), and (5.28), we can obtain the *estimation* of the number updates due to packet loss and compare it with simulations. The results are shown in Figure 5.20 and Table 5.4. The keep-alive period, i.e., the time between two consecutive keep-alive messages is distributed uniformly in the interval [15, 17] seconds. The packet loss period is the period in which no packets can be received. We set this period to be 45 sec, which guarantees that at least two consecutive keep-alive messages are lost. As expected, the simulation shows a slightly lower amount of updates than the analytical estimations. As mentioned in Section 5.3.2 and Section 5.3.3, the assumption of having a very high density network implies that the appearing or disappearing node has a maximum ith hop node degree and that no new shorter paths are generated between the appearing node and the other nodes. For low density networks, this is not true anymore. The accuracy of our analytical model therefore decreases towards low densities. The error between analytical and simulation results is 63.04% for a low density network with 0.87×10^{-5} node/m^2, but it is only 3.90% for a high density network with 5.19×10^{-5} node/m^2.

We also study the effect of different packet loss periods. We use the 61-node scenario (density of 2.11×10^{-5} node/m^2). We vary the packet loss period from 18 sec to 100 sec. The results are shown in Figure 5.21. For a packet loss period smaller than 15-20 sec, there is at most one packet lost, which implies that no updates are

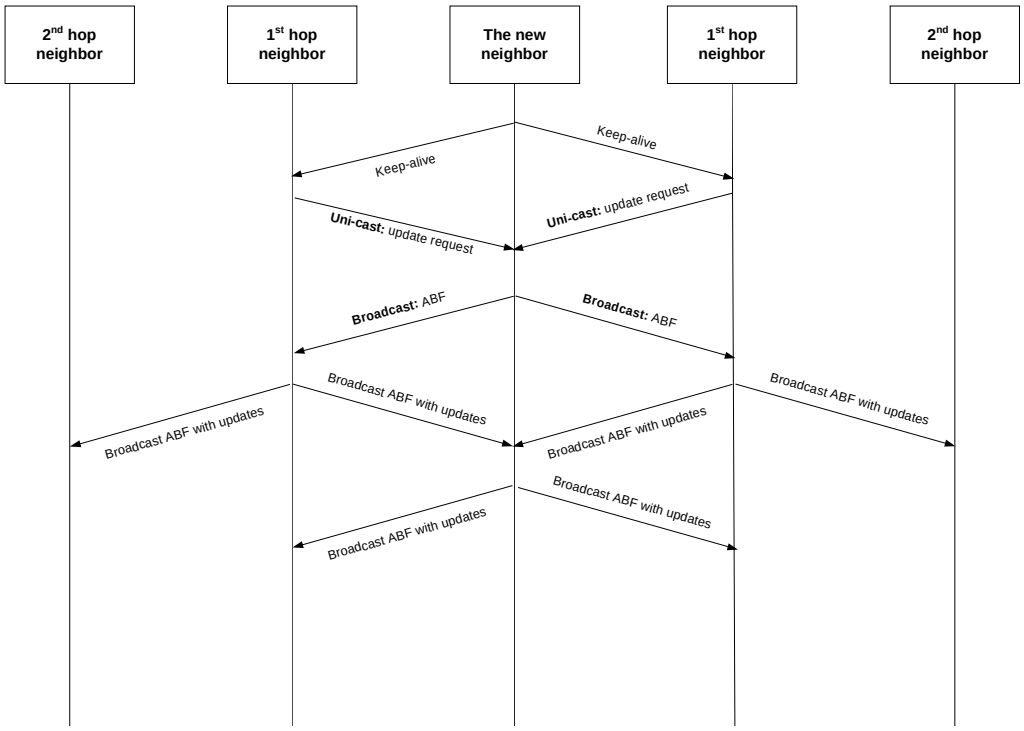


Figure 5.19: Sequence diagram when a node receives a keep-alive message from an unknown neighbor.

needed. For a packet loss period between 20 and 30 sec, two consecutive keep alive packets may have gotten lost. In this time window, the number of updates will therefore increase rapidly until it almost reaches its maximum at 30 seconds. For packets loss periods beyond 30 seconds, the number of updates only slightly increases, probably due to the fact that not all updates are done before the node reappears. If the packet loss period is larger than 45 sec, the number of extra updates is constant, because the nodes have enough time to complete the updates for disappearance within this time period.

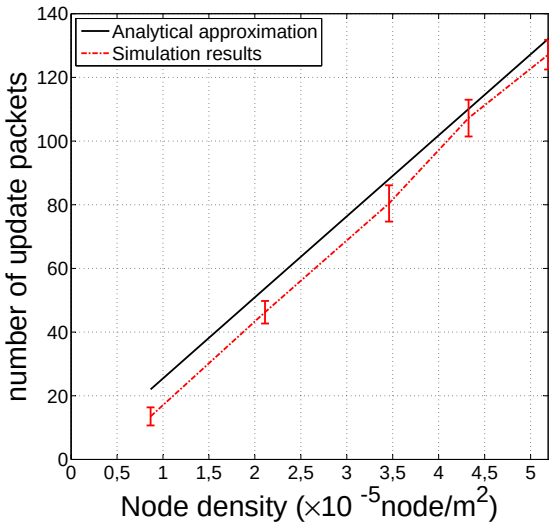


Figure 5.20: Number of updates generated due to packet loss.

5.3.5 One Moving Node

For grid structured networks, we have already shown that calculations for a moving node are rather complex, even though grid-structured networks have a highly symmetric and regular network topology. The network topologies for circular-structured networks are random and unpredictable, which complicate the calculations of the number of updates even more. In this section, we therefore only use simulation to evaluate the amount of extra traffic that is generated by a moving node. The node is moving from one spot to another in a straight line with fixed speed. The start and end point of the journey are far enough apart so that they do not share any node within d hops. This guarantees that all nodes in the query range (d hops) of the start and end position have to update their filters. This offers us the chance to compare the simulation results with analytical results in which a node disappears in one location, and reappears in a different location. This is an extreme case in the sense that the mobile node is moving extremely fast, so that the other nodes along the trajectory of the moving node will not have time to update their filters. The simulation has been done in a two-dimensional area of $4200 \times 1800\text{m}^2$. We assume that d is equal to 3. The mobile node with a 300 meters communication range is

Table 5.4: Results for packet loss in circular-structured networks.

Density ($\times 10^{-5}$ $\frac{node}{m^2}$)	Analytical Estimations	Mean Simulation Results	Confidence Interval	Error (%)
0.87	22.01	13.50	(10.66, 16.34)	63.04
2.11	53.71	48.23	(44.69, 51.77)	11.36
3.46	88.05	80.40	(74.68, 86.12)	9.51
4.33	110.06	107.20	(101.42, 112.98)	2.67
5.19	132.06	127.10	(122.45, 131.75)	3.90

set to move 2400 meters from point (900m, 900m) to (3300m, 900m). In this way, all nodes in the region can reach the mobile node within d hops and border effects are minimized.

The additional update traffic is highly related to the speed of the node and the network density given a fixed keep-alive period. To capture these relationships, we use two different network densities 8.07×10^{-6} (61 nodes in the experiment area) and 1.98×10^{-5}node/m^2 (150 nodes in the experiment area), and we let the speed vary from 0.1m/s to 20m/s. These speeds correspond to moving objects in daily life, i.e., 1m/s is the average speed of a walking adult; 5m/s is the average speed of a bicycle; 20m/s can be considered to be representative for the speed of a car. The results are shown in Figure 5.22. The figure shows that the traffic load decreases with speed. This result is not unexpected. When a node moves faster, the probability increases that its temporary neighbors do not have enough time to update their filters before the moving node has disappeared again. As can be expected, the extra network traffic increases (almost proportionally) with density, because more nodes need to update their filters when the density increases.

We also simulate the extreme case in which we use a very high speed of 24000 m/s. We assume that at this speed, nodes along the trajectory of the moving node will not have time to notice any change. This scenario can be considered as the equivalent of a node disappearing from one position and reappearing at another at the same time. The effect of the disappearance of a node is discussed in Section 5.3.2. However, the reappearance is different from that discussed in Section 5.3.3. In

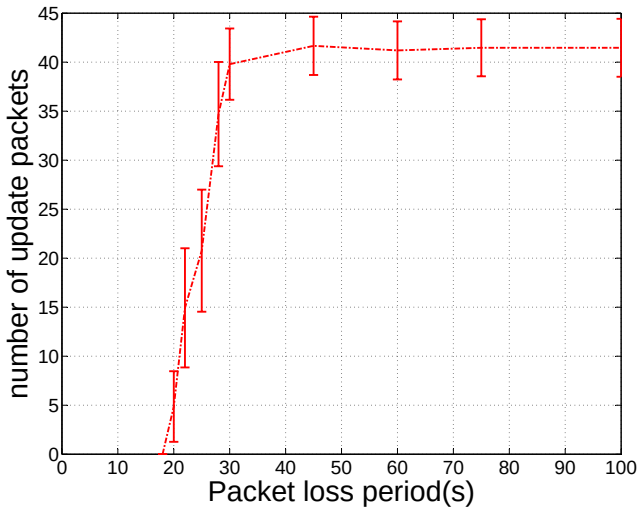


Figure 5.21: Number of updates generated due to various packet loss periods.

Section 5.3.3, we assumed that the node appears in a new environment with only its local information stored in the ABF. In this scenario, the node appears in a new environment with its ABF filled with information from its neighbors in the previous position. Therefore, the updates will take place twice. In the first round, the updates will spread the information that was still present in the ABF of the mobile node. After two continuous keep-alive periods, the mobile node notices the loss of the connection to its previous neighbors. It will clean up those neighbors from its filter, and the second round of updates will take place. We can generalize the total number of updates due to a fast moving node, $N^c_{updates\_move}$, as:

$$N^c_{updates\_move} = N^c_{updates\_nd} + 2 \cdot N^c_{updates\_na}. \tag{5.29}$$

From (5.25), (5.27), and (5.29), we can obtain the expected value of the number of updates when one node moves and compare it with the simulation results for different network densities: 8.07×10^{-6}, 1.98×10^{-5}, 2.38×10^{-5}, 3.17×10^{-5}, and 3.97×10^{-5}nodes/m^2. The results are shown in Figure 5.23. Again, the analytical expected number of updates overestimates the amount of extra traffic when the network density is low, because it overestimates the connectivity and thus the number of nodes that needs to be updated. For higher network densities, our analytical model

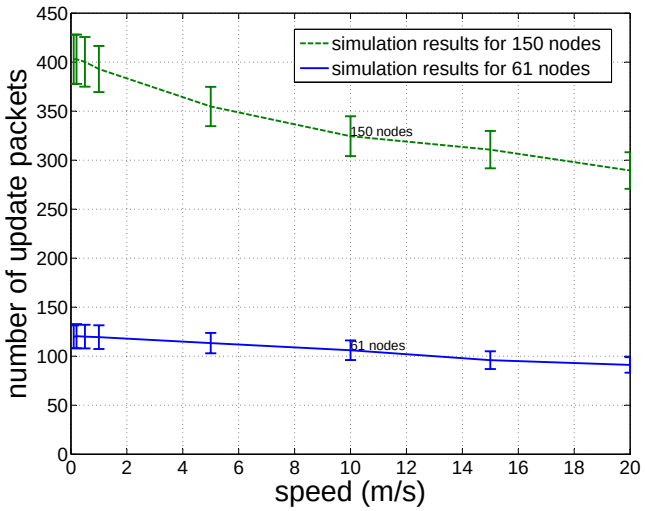


Figure 5.22: Number of updates generated when one node moving in different speed in various network densities.

becomes more accurate. In the high density scenarios with 2.38×10^{-5}, 3.17×10^{-5}, and 3.97×10^{-5}node/m^2, the analytical results are within the confidence interval of the simulation results.

5.3.6 Summary

Updates due to dynamic connectivity happen whenever a shortest path between a pair of nodes has been modified. We have obtained the expected number of additional broadcasts in analytical expressions for the circular structure, with the assumption of high density networks. Simulations have been done for comparison. From this, we conclude that the analytical expressions are indeed quite accurate for high-density networks. This fits our hypothesis. For low densities, the analytical model overestimates the number of updates, because, among others, it then overestimates the i-hop node degree.

Since the proposed protocol automatically duplicates its own context sources to all lower layers of the ABF, the appearance of a new node can be handled in a single pass of advertisements. No advertisements have to go up and down to propagate

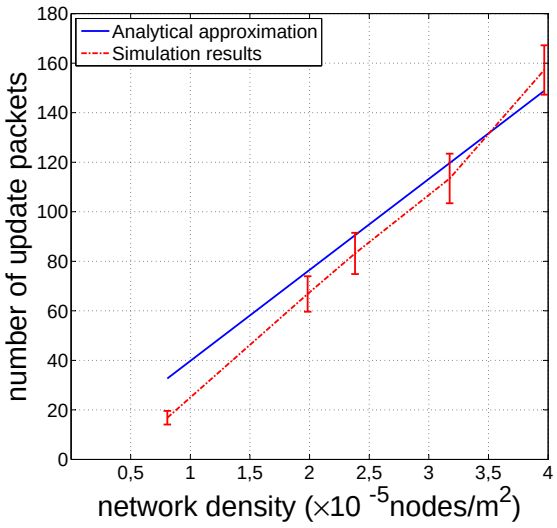


Figure 5.23: Number of updates generated when one node moves with the extremely high speed of 24000 m/s in various network densities. In this case, no update is generated during the move. It is equivalent of the case that a node disappears from one position and reappears at another that no node knows it.

the availability of indirect paths to the new node into the lower layers of the ABFs of the other nodes. However, in the case of removal of a node, multiple passes are needed to remove its representation completely from all layers of the ABFs. Removal of context sources has to be done layer by layer, as there is no equivalent of duplication in this case. Therefore, removing a node generates more traffic than adding a node.

We studied a simple mobile case of one node moving in a straight line in a circular structured network. For increasing speed of the moving node, less traffic is generated in total. However, we believe that in real world situations, network density has a larger effect on the number of generated updates in total than the speed of the mobile node. In the extreme case where a node moves at very high speed, the analytical model fits the simulation results quite well for high density networks.

From this analysis, we can anticipate the network behavior in terms of updates

when nodes are moving in the network, and when densities vary. There are more updates required in high density networks, whereas there are fewer updates generated when nodes move faster. Of course, the keep-alive period is also an important aspect in this respect. The shorter the keep-alive period is, the faster nodes notice the change in the network. As a results, more updates are generated. On the other hand, large keep-alive periods generate fewer updates, but the information is not up to date. When queries are sent out often, there will then be a slightly higher chance of false positives and false negatives. The choice of the keep-alive period depends on the network situation and the requirements. The network administrator should choose which is more important: the amount of traffic or the discovery efficiency.

5.4 Comparison between three protocols

In Chapter 4, we compared the performance of `Ahoy` with that of the reactive and proactive protocol in terms of overhead traffic (including frequent updates and false positive queries). In that chapter, we concluded that the protocol choice should depend on the ratio between query and advertisement rates. When queries, compared with advertisements, are occasionally sent, the reactive protocol performs the best. When queries are sent very frequently, the proactive protocol performs the best. `Ahoy` performs the best in between these two extreme cases. Dependent on the different network settings, the range of ratio's in which `Ahoy` performs best varies. For example, it is between about 0.1 and 10^7, when $s = 4$ and $d = 5$ (see Section 4.4.1).

In dynamic networks, `Ahoy` and the proactive protocol need to broadcast frequent advertisements to update changes in the network. The reactive protocol is not influenced in this respect, because it does not broadcast any advertisement. For obtaining the same performance in the dynamic cases, i.e., to keep the same ratio of queries and advertisements, the reactive protocol can send more frequent queries in dynamic networks than in static ones. The difference with the static case is getting smaller when nodes move fast, because fewer extra updates are then generated (see Section 5.3.5).

The overhead traffic generated by `Ahoy` and by the proactive protocol in dynamic networks is hard to compare. With the proactive protocols, such as CDS[26], nodes generally record the existing context information types and their addresses. It is the

task of the underlying routing protocols to locate the position of the requested context information types. They remove the disappearing context information types locally when they are not refreshed for a certain time interval. No further announcement is sent to the network. On the other hand, `Ahoy` takes over part of the effort from routing protocols and can provide possible discovery directions. It keeps routing information updated whenever there is any change. Especially, in dynamic networks, `Ahoy` spends much effort to update such routing information, e.g., advertisement loops are generated when context information types are removed. It is not fair to compare directly the overhead traffic generated by `Ahoy` and the proactive protocol any more in such an environment.

Chapter 6

Vulnerability Analysis

Security is one of the major concerns in telecommunication and computer networks, especially in mobile and wireless environments. General security challenges in ad-hoc networks are due to unsecured wireless links, limited resources, large and dense networks, lack of fixed infrastructures, unknown topology, and physical attacks [84].

From the literature, see e.g., [4, 81], attacks in ad-hoc networks can be classified into two categories: passive and active attacks. Passive attacks do not interfere with networks directly. One typical behavior is network eavesdropping. In contrast, active attacks influence networks directly, such as active interfering, and denial of service (DoS) attack. Active interfering attacks try to interfere with the message packets, including packet dropping, modification, and packet replaying. DoS attacks attempt to make context sources unavailable for the intended users, e.g., by flooding packets.

These different attacks cause a variety of damage against security attributes. [4, 81] define a few basic security attributes: *confidentiality*, *integrity*, *availability*, *authentication*, and *non-repudiation*. For discovery protocols in MANETs, damage can be categorized into three classes: privacy intrusion, lower discovery efficiency, and network jamming. Privacy of nodes is intruded, if confidentiality of those nodes is violated. The discovery efficiency gets lower and networks are sometimes jammed, if integrity and availability of nodes are disrupted. Since the original `Ahoy` does not have authentication requirements and needs to assure the originator of a message, we do not consider the security attributes of authentication and non-repudiation. In this chapter, we discuss the damage in detail and compare their effects on three types of protocols: `Ahoy`, the proactive, and reactive protocols. In the remainder of

the chapter, we refer to them as "the three protocols". In case of serious risks, we propose security countermeasures to enhance the network security in `Ahoy`.

This chapter is structured as follows. We briefly introduce the attacks that we focus on, and the damage they generate in Section 6.1, and 6.2, respectively. We then describe qualitatively to which extent the three protocols can be damaged by the attacks, and how they compare in this respect. We compare privacy intrusion for the three protocols in Section 6.3, discovery efficiency in Section 6.4, and network jamming in Section 6.5. In Section 6.6 we propose possible countermeasures. Finally, we conclude the chapter in Section 6.7.

6.1 Summary of Attacks

Vulnerability of a discovery protocol is any software flaw that leaves a discovery protocol open for potential exploitation, and an *attack* towards a discovery protocol "is an attempt to bypass the security controls" of such a protocol. "The success of an attack depends on the vulnerability of the system and the effectiveness of existing countermeasures" [4]. In this chapter, we analyze various attacks to the discovery protocols, especially regarding the exchanged information during the discovery phase, and study their impacts on the discovery protocols. We address the performance of a discovery protocol, when adversaries attempt to eavesdrop, delete, modify, replay, and flood exchanged information. We do not consider attacks towards other communication protocols and other layers, such as wormhole and blackhole on the network layer, and traffic analysis, monitoring, and WEP weaknesses on the data link layer. The attacks that we consider, are listed as follows.

- **Network eavesdropping.** Adversaries sniff the exchanged packets, but do not influence them directly. They learn the parameter settings and the contents of the packets. The eavesdropping itself does not damage the performance of the network. However, it intrudes the privacy of the nodes in the network. Adversaries intend to gather specific information from the different packets as is shown in Table 6.1.

- **Packet dropping.** Adversaries refrain from processing and forwarding the packets. The nodes that are supposed to receive the packets do not receive

Table 6.1: Information gathered from different packets by network eavesdropping.

Packets	Information Gathered
Advertisement	Which information is available where in the network?
Query	Who is looking for which information?
Reply	Who is providing which information to whom?

them. Depending on which packets the adversaries drop, the impact from this attack is different, as shown in Table 6.2.

Table 6.2: Impact on dropping different types of packets.

Packets	Impact
Advertisement	Information in some part of the network is outdated.
Query	Information in some part of the network can not be discovered.
Reply	Information in some part of the network can not be reached.
All packets	Adversaries are out of reach for other nodes.

- **Modification.** After capturing packets, adversaries may modify contents of the packets and forward them further into the network. Nodes containing false information behave erroneously. The major impact include lower level of context availability, longer searching time for required information, and extra traffic to discover non-existing information.

- **Packet replaying**. Adversaries keep on replaying outdated packets to the network rather than sending the updated packets. Although the impact is not as severe as the ones from modification attacks, they are of the same kind.

- **Packet flooding.** Adversaries send massive amounts of packets at high rate through the network. This is one of the Denial of Services (DoS) attacks which attempts to make the requested information unavailable for the intended users.

No matter which packets adversaries generate, the major objective is to jam parts of the network, and in some cases even to break down the entire network.

6.2 Damage from the attacks

The basic security attributes with respect to a discovery protocol are:

- **Confidentiality** ensures that unauthorized entities cannot overhear certain information.
- **Integrity** ensures that exchanged information is not modified nor corrupted.
- **Availability** ensures that intended context information is accessible.

When adversaries successfully attack the discovery protocol to violate the above security attributes, other nodes and the network are damaged in the following three aspects: privacy intrusion, lower discovery efficiency, and network jamming.

- **Privacy intrusion.** Adversaries try to violate *confidentiality* and sniff information, which is exchanged in the network, and which is not public. There are two kinds of networks in terms of privacy: public and private. Public networks are open to everyone. The information of such a network can be shared with all the parties. Private networks have a limited number of participants. Information is restricted to these participants. When adversaries try to capture and read the information in private networks, they invade the privacy of the participants of the network. This type of damage is generally caused from network eavesdropping attacks.
- **Lower discovery efficiency.** When correct (and updated) information is published in the network, a proper discovery protocol can locate the required information in the network if it exists. When adversaries disorder this information or the transmission schedule, nodes can not find the right information, or can only find the right information after some delay. In this case, *availability* and *integrity* are violated. This type of damage can be caused by packet dropping attacks, modification attacks, and replaying attacks.

- **Network jamming.** Some attacks towards *availability* and *integrity* can lead to network jamming, which may result in a (partial) network breakdown. Packets cannot pass through this (part of) network. It can also cause a delay in the discovery of information. This type of damage is mainly evoked by packet flooding attacks. Adversaries simply release large amounts of redundant packets into (parts of) the network. Network jamming can also be induced by attacks like packet dropping, modification, and replaying. When false information is propagated through the network, wrong queries can be sent to the wrong nodes. When adversaries cooperate, they can target certain node(s) and exhaust the network in that area.

We summarize the damage and their related attacks in Table 6.3.

Table 6.3: Damage from various attacks in the network.

Damage	**Attacks**	**Violated Security Attributes**
Privacy intrusion	network eavesdropping	confidentiality
Lower discovery efficiency	packet dropping, modification, replaying	availability, integrity
Network jamming	packet flooding, packet dropping, modification, replaying	availability, integrity

In the following three sections, we investigate all three classes of damage on `Ahoy`, as well as on the proactive, and the reactive protocol in detail. We compare the vulnerability of the three types of protocols. For a fair comparison, we assume that the advertisements in the proactive protocol and the queries in the reactive protocol are broadcasted within the same discovery range as in `Ahoy`, i.e., within d hops. We do not weight the consequences of each class of damage here, because the amount of damage highly depends on the different scenarios and perspectives. For example, in a private network, end users might consider privacy the most important issue; while network administrators focus more on the overall network traffic conditions.

6.3 Privacy Intrusion

The privacy of network participants is intruded when the exchanged information is eavesdropped by adversaries. `Ahoy`, the proactive, and the reactive protocols, expose different information in their advertisement, query, and reply packets. Here, we assess two aspects for three different protocols. First, the rate at which packets can be captured by the adversaries. This rate depends on how often and to which extent packets are exchanged. Second, the amount of information adversaries can obtain from a specific packet. This depends on how accessible the information is, i.e., how well the information in a packet is secured against adversaries.

6.3.1 Sniffing of advertisement packets

In `Ahoy`, context advertisement packets contain the hashed context information types of nodes within the range of d hops. These packets are sent whenever there is any change on the existence of the information in reach. We do not consider keep-alive packets here, since there is no conceptual information contained in such packets. Adversaries can learn the following knowledge directly from an advertisement:

- Parameter settings of the ABF, including the width w and the depth d, which is also the maximum number of hops for context discovery.
- Information regarding context density distribution in the network. ABF utilizes 0 and 1 to represent the existence of context information types. The more 1s' are located in the filter, the more information types are available by the current node.

Adversaries can not easily retrieve all exchanged information from the packets, since only information a few hops away is captured. Also, the context information is coded by hash functions. Theoretically, if we use one-way hash functions, it is unlikely to retrieve the original information from the hash results [72]. However, if the context information types are also public and standardized, adversaries can obtain the hash results by hashing every type and compare the hash results. To decode an ABF with multiple types, adversaries need to not only hash each single type, but also the combinations of the types. Thus, it will cost adversaries extra effort to existing information in the network.

If `Ahoy` uses standard context information types and hash functions, adversaries can check whether specific context information is available somewhere. As discussed in Chapter 4, the false positive probability increases with the number of hops. Adversaries can only test with quite good accuracy whether specific context information types are located in their direct neighbors. This information can be used later, if adversaries would like to modify the content of packets or jam the traffic towards some nodes.

On the contrary, the proactive protocol broadcasts the original information to all the nodes within d hops. Once such a packet is captured, adversaries can easily locate the position of the related information within a certain scope. In the worst case, adversaries will know the locations of all the information in the network.

The reactive protocol does not exchange any information beforehand. As a result, there is nothing to be eavesdropped in this phase.

We summarize the two privacy aspects we considered when adversaries eavesdrop advertisement packets into Table 6.4.

Table 6.4: The vulnerability of three protocols when adversaries sniff advertisement packets.

	Ahoy	**Proactive**	**Reactive**
Packet propagation range	all nodes in d hops	all nodes in d hops	none
Content format	ABFs	original information	none
Consequences	only regional information exposed, hard to retrieve, but can check the availability	some/all information exposed	none

It is obvious that reactive protocols protect the "where is what" data best, since there are no advertisement packets exchanged in reactive protocols. In both `Ahoy` and the proactive protocol, advertisements are only propagated within a limited range (d hops). However, in the proactive protocols, all advertisement information

is public to every one, while in `Ahoy` information is hashed. Adversaries can obtain advertisement information, but it is not easy for them to obtain the exact location of the available context information. In general, `Ahoy` therefore performs only slightly worse than reactive protocols, but it performs better than the proactive protocols.

6.3.2 Sniffing of query packets

Whenever a node looks for certain context information, it sends out a query. In the design of `Ahoy`, queries can be sent out as either original text strings or BFs. If queries are sent in the string format, adversaries can learn exactly who is looking for what information from the sniffed query packet. If queries are hashed into BFs before sending out, adversaries cannot easily decode the queried information from BFs.

One straightforward way to decode queried information is by hashing arbitrary context information types, and comparing the results with the query BFs. The amount of work to decode highly depends on the number of context information types in the network. On average, adversaries need to perform the total number of context information types divided by 2 times calculations to decode one query BF. Note that this, in general, requires much less effort than finding all the information from ABFs (discussed in Section 6.3.1), which bits may be shared by several context information types that result in false positives. Without decoding query BFs, adversaries only know who is looking for information, but not which information. In this respect, we prefer to send queries as hashed BFs rather than text strings. However, there are also other issues that we have to consider. In Section 6.4.1, we will show that hashed BFs also have drawbacks.

The exposure to adversaries also depends on the number of nodes that need to forward a query. If more nodes are involved in querying, the probability increases that an adversary sniffs a query. The number of nodes that forward queries depends on the chosen query method. There are two possible query methods in `Ahoy`: parallel and sequential.

- In the parallel query method, a query is forwarded to all neighbors with which ABF a match is found. The query is forwarded until the node is reached that contains the queried information or until the query has traveled d hops.

- In the sequential query method, a query is first sent along one path. If a node along this path contains the queried information, querying is stopped. Otherwise, it is sent along other paths until the context source is found. In the best case, only nodes along one query path need to propagate the query message. In the worst case, all paths that may lead to a match are searched, just as in the parallel method.

In general, fewer nodes are therefore used in the sequential method. The drawback of the sequential method, however, is that it in general will also take more time to find the context source. In that respect a choice has to be made between speed on the one hand, and the amount of generated traffic and level of security on the other hand.

Both the proactive and reactive protocols send queries in original formats, such as text string, XML, etc. In the proactive protocol, only one query is sent to the exact destination, while in the reactive protocol, queries are broadcasted to all the nodes within d hops. Obviously, there is a higher probability of capturing a query packet in the reactive protocol than in the proactive protocol. We summarize above discussions into Table 6.5.

Table 6.5: The vulnerability of three protocols when adversaries sniff query packets.

	Ahoy	**Proactive**	**Reactive**
Packet propagation range	selective nodes in d hops in maximal	one node and related nodes in the path	all the nodes in d hops
Content format	BFs / text strings	original information	original information
Consequences	selective queries, possibility to hash them	only exposed to the nodes along one path	all information exposed

`Ahoy` sends out fewer queries than the reactive protocol due to the feature of "selective querying". However, in general, it sends out more queries than the proactive protocol. Only in the best case of sequential querying, nodes along one path need to

propagate queries, as is the case in the proactive scenario. Especially in scenario's in which each context type is scarcely available in the network, the "selective querying" of `Ahoy` will pay off. In that case, the performance of `Ahoy` will be comparable to the proactive approach, whereas it is likely to perform significantly better than reactive protocols, in which adversaries can easily obtain query information.

The three protocols expose the same amount of information once a query packet is captured. However, in `Ahoy` there is the option to protect the queried information by sending queries in BF format, and thus force adversaries to use a considerable amount of effort to understand which information a node is looking for.

6.3.3 Sniffing of reply packets

Reply packets are composed of the related query packet identification and the IP address of the replying node that has the requested context information. `Ahoy`, the proactive, and the reactive protocols have the same message format. Reply packets are sent by a node when it discovers that it contains the queried information. Adversaries can obtain the same amount of information about who is providing information in all three protocols.

6.3.4 Summary

`Ahoy` exposes much less information than the proactive protocol and slightly more than the reactive protocol in the context advertisement phase. It generates much less queries to be sniffed than the reactive protocol and slightly more than the proactive protocol in the context query phase.

`Ahoy` has an option to hash the queried information so that adversaries cannot obtain it easily. In the context reply phase, all three protocols perform the same in terms of vulnerability.

We do not evaluate the importance of information here. It highly depends on the different scenarios and perspectives. "Who has what?" (learn from advertisements) and "Who is looking for what?" (learn from queries) are considered equally important. `Ahoy` performs reasonably well in terms of privacy compared to the conventional protocols. `Ahoy` can protect both types of information by using Bloom

filters. Adversaries can decode original information from hash results. However, it requires a lot of extra effort from adversaries to do so. It does not damage the nodes themselves if one or more adversaries eavesdrop packets, but the sniffed information might be used for other attacks, such as packet modification and flooding.

One solution to protect data for all three protocols is encryption. Section 6.6 addresses this in more detail.

6.4 Lower Discovery Efficiency

When adversaries release incorrect information in the network, nodes can not discover the right information, or only after some delay. Adversaries can modify the contents of packets or send incorrect (or outdated) information themselves. They can also eliminate updates by simply dropping entire packets, or delay the updates by replaying old messages. Network eavesdropping does not have a direct influence on a lower discovery efficiency, but it can provide the adversaries with necessary information to modify packets.

In the following subsections, we analyze the influence of modification, packet dropping, and replaying on the different types of packets, and we propose possible intrusion prevention solutions.

6.4.1 Modification

Three kinds of packets can be modified by adversaries: advertisement, query, and reply packets. Modified packets are (partially) distributed over a d-hop range, if the discovery range is restricted to d hops for all three protocols. How many nodes will be affected, depends on the protocol and the location of the adversary node. The closer the adversary node is to the node that sends the packet, the larger the area is, through which the modified packet is propagated. In case of advertisement, the adversary node itself can also send false information, which will be distributed over all the nodes within d hops in both `Ahoy` and the proactive protocol. In general, modification of advertisement packets reduces the information availability directly, and thus requested context information types either cannot be found or might take longer time to be found when queries are sent sequentially. If query packets are

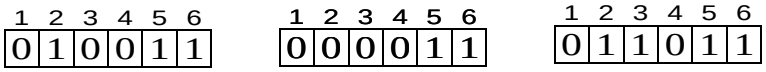
modified, requested information cannot be found. Adversaries can also fake the replies, which leads to incorrect connections, no connection at all, or can lead to an overload of traffic around adversaries and their neighboring nodes.

The differences between the three protocols narrow down to differences in packet propagation frequencies and ranges, and content formats. We discuss the impact of modifying different packets in detail in the following.

Modify context advertisement packets. When adversaries modify information, nodes in the advertisement propagation range are affected. Advertisements are sent to all nodes d hops away in `Ahoy` and the proactive protocol, and there is no advertisement in the reactive protocol. The same number of nodes are thus affected in both `Ahoy` and the proactive protocol, while no node is affected in the reactive protocol.

There are also differences in format. `Ahoy` presents all advertisements in ABFs, while the proactive protocol utilizes the original text strings. Adversaries can change the bits (in `Ahoy`) or the letters (in the proactive protocol) simply without understanding what is presented in the advertisement. In `Ahoy`, one bit can be shared by multiple context information types. Changing a bit from 1 to 0 can remove one or more context information from the filter; changing a bit from 0 to 1 can add one or more context information in the filter. For example, we have the following filter in Figure 6.1(a). We assume "temperature" is coded into the bit position "2" and "6"; "printer" is coded into the bit position "2" and "5". When we change the bit position "2" from 1 to 0, as is shown in Figure 6.1(b), both the "temperature" and "printer" information types are removed. This will lead to the decision not to check or forward queries that request for that information (false negatives). When we change the bit position "3" from 0 to 1, as is shown in Figure 6.1(c), it adds "beamer" (the bit position "2" and "3"), "music player" (the bit position "3" and "5"), and "VoIP" (the bit position "3" and "6"). This will create extra queries that are forwarded to nodes that do not have that information (false positives).

Adversaries can modify advertisements arbitrarily or specifically. Arbitrary modification revises the contents of advertisements randomly. In the proactive protocol, this would mean that letters are arbitrarily changed in an advertisement. In English, there is a slight chance that two or more words have the



(a) Original (b) Modify the bit position "2" (c) Modify the bit position "3"

Figure 6.1: A filter when its bit position "2" or "3" is modified.

same length and only differ by one letter. The probability of multiple context information types being removed or added at once, is smaller than in `Ahoy`. The strength of `Ahoy` is that it can compress information. However, this also makes it vulnerable, because slight modifications can lead to relatively large losses of information.

If adversaries specifically modify advertisement contents, they would like to add or remove some specific "wrong" information into the advertisement packets. This can be easily done in the proactive protocol. In `Ahoy`, this is only possible if adversaries know the applied hash functions. As was discussed in Section 6.3.1, it will take a considerable amount of effort for the adversaries to learn the hash functions. In this sense, `Ahoy` is therefore less vulnerable than the proactive protocol.

There is another field in the advertisement packets of `Ahoy`, which is called the *generation ID*, GID. If the GID is modified, there might be extra update requests and advertisements due to the maintenance principles defined in `Ahoy` (see Chapter 3). There is no specific definition of this fields in the proactive protocol. We can, however, imagine that the proactive protocol also has a similar field to identify the advertisement packets and that it applies a similar mechanism to request for updates.

We summarize the above discussion in Table 6.6. The reactive protocol performs the best in the advertisement phase. The same number of nodes are influenced in `Ahoy` and the proactive protocol. Modifying the GID field in `Ahoy` might cause some unnecessary updates, but the effects are minor. The

proactive protocol might be affected in the same way, if it also applies a similar mechanism. The modification of the advertised information itself is more damaging. When adversaries modify information arbitrarily, the discovery efficiency in the infected region is lower for `Ahoy` than for the proactive protocol. Specific information, however, can be more easily modified in the proactive protocol than in `Ahoy`. The vulnerability of `Ahoy` and the proactive protocol depends on several things, such as advertisement and query rate, d, the size of the network, and whether or not the hash functions in `Ahoy` are standard and public.

Table 6.6: The vulnerability of three protocols when adversaries modify advertisement packets.

	Ahoy	**Proactive**	**Reactive**
Packet propagation range	all nodes in d hops	all nodes in d hops	none
Content format	ABFs	original information	none
Consequence	Partially influenced with high chance of false positives and negatives, because slight modification may lead to large changes. Intentional modifications only possible when hash functions are known.	Part of or the entire network, but less false positives and negatives than in `Ahoy`. Intentional modification can be easily done.	none

Generally, the effect is limited when only one adversary modifies advertisement packets. Most likely, nodes can reach the appropriate context sources via other paths, which are not affected. Moreover, if adversaries only modify advertisement packets once, the "wrong" information will be replaced by the correct one when the ABFs are updated. If adversaries continue modifying advertisements, the "wrong" information is also kept in the updates. The infected region will then continuously be malfunctioning. In addition, it will

take a lot of effort for `Ahoy` to remove "wrong" information layer by layer, while the proactive protocol can simply remove it at once. In Section 5.3.2, we discussed the process and the number of updates needed to remove one information in an `Ahoy` network. In the worst case, several adversaries coordinate attacks in a region, and a node in that region might completely be shut off from its surroundings.

Adversaries thus only need little effort to modify advertisement packets, whereas they can severely affect nodes. However, even in the worst case, the overall effect on the network will probably be limited, because the information in the network is distributed in a very decentralized way. Contrary to centralized protocols, the damage that is done to one or even several nodes, does not necessarily have an effect on the other nodes in the network.

This strength of a fully distributed ad-hoc network is also a weakness, because adversaries can quite easily damage it without being noticed. Without knowing what information is exactly included or accessible from others, it is hard to detect this kind of attacks by only reading the contents. For the proactive protocol, when the density of adversaries is low compared with that of the other nodes, it is possible to detect adversaries by comparing advertisement contents from multiple neighbors, for example through Byzantine agreement [48]. When there are many adversaries in one region, it is difficult to recognize which nodes provide the correct information. For `Ahoy`, comparing ABFs from neighbors will not directly lead to the detection of adversaries, since information in ABFs is aggregated. However, it is possible to check for inconsistencies in the ABFs to detect an attack. According to our study in Chapter 4, the number of false positive queries can be restricted if the width of the filter w is larger than the product of the number of hash functions b and the maximum numbers of context information types in reach. We need to define a large enough w to achieve a small number of false positives. Such a filter should have a fair ratio of bits that are set to 0 and 1. Furthermore, all the information that is in the upper layers, should also be duplicated in the lower layer (see Section 3.3.3). We propose to apply rules to check the proportions of 0s' and 1s' in the filter and to check for duplication of the information in the lower layers. This issue is further addressed in Section 6.6.4. Finally, modifications can be detected

by using the Michael algorithm (the message integrity code) [23], which will require larger packet sizes and more computations, as is addressed in detail in Section 6.6.

Modify query packets. In the proactive protocol, queries are sent to specific nodes along one path. In the reactive protocol, queries are sent to all the nodes within d hops. In the proactive protocol, only few nodes will therefore be affected by a modified query packet, while many more nodes will be affected in the reactive protocol. In `Ahoy`, queries are sent either to one possible destination or to all nodes that possibly could contain the information. The number of nodes that are queried depends on the query method, as was discussed in Section 6.3.2. Most of the time, `Ahoy` has a slightly larger, but still comparable impact area as the proactive protocol. In general, the reactive protocol has a much higher probability to be attacked than the two other protocols.

An `Ahoy` query packet is composed of query identification (QID), query message, a sender address, and time-to-live. QID is the unique identification for each query. The query message is the information which the query node is looking for, and can be sent in a text string or BF format. Time-to-live indicates the number of hops a query can be propagated further. Sender address is the IP address of the query initiator. A query packet in the proactive or reactive protocol can contain the same information with the exception that query messages are always in original formats. Modifying QIDs, query messages, or sender addresses can prevent the query node from finding the correct information. As a result, no information or wrong information is found. If there exist other paths to the requested context source, which do not pass the adversaries, it is still possible to find the requested context information. In this respect, the reactive protocol and the parallel query method in `Ahoy` are more likely to find that information faster than the proactive protocol or the sequential query method in `Ahoy`.

Time-to-live is a specific parameter that restricts the propagation range. Modifying the time-to-live parameter influences the query range. If it is modified to a smaller number, the discovery range shrinks and the probability to find the required information reduces. If it is modified to a large number, the

query may be forwarded a large number of hops. Certainly, there is a higher probability of obtaining the required information in a larger discovery range. However, we do not prefer to access information very far away, because in that case more query traffic is generated, and there is a higher false positive probability.

If more adversaries are located in a specific region, there is a higher probability of having false queries in that region. In the worst case, a targeted attack occurs in which all query packets from a specific sender are modified. During a targeted attack, the specific sender can not find any information, because it is isolated by the adversaries. Such an attack can occur in all three protocols, but is rather difficult in a mobile network, because it requires adversaries to stay in the query range of the targeted node.

In a reactive protocol, the probability that queries are incorrectly formulated is relatively high, because many queries are sent. However, the probability of being affected is not the only concern. A single attack (with low probability) can still be quite harmful. When adversaries capture both advertisements and queries, information does not only become unreachable, but adversaries can also direct all the query traffic to a targeted nodes and exhaust that node. This can be done by modifying queries, such that they look for all context information that is advertised by the targeted node. This is not possible in the reactive protocol, because there is no advertisement and it is not known which context information is located where. Similarly, adversaries cannot target specific nodes in `Ahoy`, when queries are sent out as strings, because adversaries cannot match the query context information with that in the advertised ABFs (suppose using perfect one-way hash functions). If the queries, however, are sent out as BFs, the adversaries can easily set the same bit positions as in the ABFs from the target nodes. In that case, the queries are forwarded to the targets and the targets get exhausted. The first node which is exhausted will be the adversary itself and some of the nodes nearby it. This can cause (rather large) negative effect. If queries are not encrypted, using the BF format can provide some level of protection over queried contents than using the text format. However, if one of the main security requirements is to prevent modifying contents of exchanged packets by adversaries, we suggest to define text strings

instead of BFs as query format. We summarize the above discussions into Table 6.7.

Table 6.7: The vulneralibility of three protocols when adversaries modify query packets.

	Ahoy	**Proactive**	**Reactive**
Packet propagation range	selective nodes in d hops	one node and related nodes in the path	all nodes in d hops
Content format	ABFs / text strings	original information	original information
Consequence	selective queries, much fewer nodes affected than in the reactive protocol, unreachable queried information through adversaries, limited (when using text strings) targeted attacks, discover faster when affected using parallel method.	affect to the nodes along one path, unreachable queried information, targeted attacks.	most nodes are affected among three protocols, unreachable queried information, limited targeted attacks, discover faster when affected.

The Michael algorithm [23] can be applied to detect modifications, as is addressed in detail in Section 6.6. Furthermore, receivers can detect the attacks if the time-to-live information is modified to a number larger than d. It is always an option to encrypt the whole query packet for all three protocols, as will be discussed in detail in Section 6.6. Additionally, we can apply a rule for `Ahoy` in which we limit the time-to-live parameter in order to restrict the influence area. This will also be discussed in Section 6.6.

Modify reply packets. Reply packets consist of QID, and IP address. Changing any of those fields will result in incorrect connections or no connections with the requesting node. It reduces context information availability and requires a

longer discovery duration if additional queries are generated. If there are more adversaries or more frequent reply packets modified, the probability to set up a correct connection to the right context source will decrease. A targeted attack can be performed in all three protocols. This is done by modifying replies from one specific node. The impact of this attack is that the targeted node can not provide all requested context information. If adversaries are located nearby the targeted node, they can block more replies. If the targeted node is only surrounded by adversaries, it cannot be used as a context source. Of course, adversaries can totally isolate the targeted node by modifying all ingoing and outgoing packets. But this is not the most efficient way to isolate a node. An easier way can be by dropping all the packets or simply interfering with the radio frequency of the targeted node.

All three protocols perform the same towards this attack. We can apply the Michael algorithm to detect the attacks. Besides, we can also monitor QID and drop the duplicated packets with the same ID which we will address further in Section 6.6.4.

Summary. In the advertisement phase, the reactive protocol performs the best. In the query phase, the chances for an attack are smallest for the proactive protocol, but the proactive protocol is quite vulnerable against targeted attacks. All three protocol perform the same in the reply phase. In all cases, `Ahoy` performs as well as or only slightly worse than the best protocol. Which protocol performs overall the best depends on the network structure, and advertisement and query rates. However, in general, `Ahoy` definitely does not perform worse than the other two protocols. Encryption is always an effective but expensive solution against modification attacks. The Michael algorithm can be used to detect modification attacks, we address this in more detail in Section 6.6.

6.4.2 Packet dropping

Adversaries can drop advertisement packets to keep information outdated, and can drop query and reply packets to prevent nodes from finding the required information without delay. Similar as for the modification attack, targeted attacks can be performed here. Adversaries can target one node, and drop all the packets from this node. If the targeted node is only surrounded by adversaries, it will be isolated from other nodes in the network. As we have already mentioned, such an attack requires the cooperation of several adversaries, and is not easy to maintain. Whenever a normal node moves into the communication range of the targeted node, a path to the other part of the network can be established from this node, and the isolation is lifted.

For the three discovery phases, the effects of packet dropping can be described as follows:

Drop context advertisement packets. In `Ahoy` and the proactive protocol, advertisements are forwarded to nodes within d hops, while no advertisements are forwarded in the reactive protocol. When adversaries drop an advertisement, only a part of the network is influenced by the attack. Especially, when the network density is high, nodes can obtain the updates through other paths. This attack has absolutely no impact on the performance of the reactive protocol, but may cause one or couple of context sources unreachable or increase the discovery duration when the proactive protocol or `Ahoy` is applied.

Drop query packets. Queries are sent to selected nodes along multiple paths in `Ahoy`, to nodes along one path in the proactive protocol, and to all nodes (within the query range) in the reactive protocol. It is most likely that the querying duration is extended in the proactive protocol and in the sequential query method of `Ahoy`, because once the querying nodes notice there is no reply, they try to query a different path. On the other hand the query duration stays the same in the parallel querying method of `Ahoy` and in the reactive protocol. Multiple paths are queried simultaneously in these protocols, which implies that dropping of query packets along one path does not influence the discovery along the other paths.

Drop reply packets. If adversaries drop reply packets, the node that sent the

query cannot discover the requested information via that path. If there exist other paths which do not pass adversaries, the node that sent the query can still obtain the requested information from corresponding context sources. To apply different query methods results in different discovery duration here. Similar as the case of dropping query packets, the sequential query method of `Ahoy` and the proactive protocol requires longer discovery duration than the parallel method of `Ahoy` and the reactive protocol.

Summary.

In general, adversaries need to spend little effort to make packets dropped in the network. On the other hand, packet dropping has a very limited impact on the context discovery, especially in high density networks where there are multiple paths between two nodes, nodes can then retrieve information via paths that are not affected by adversaries. However, the negative impact increases when there are more adversaries that can drop packets. In that case, there is a higher risk that all paths between two nodes are blocked by adversaries, and that some nodes in the network cannot discover and share information between each other any more.

This type of attack is very difficult to be detected. If a node does not receive advertisements from a neighbor for a while, it might conclude that the availabilities of context information types reachable from this neighbor have not been changed. For query and reply packets, a node might not even notice if some or all of them are dropped, because queries and replies are not mandatory packets that a node must receive periodically.

`Ahoy` performs slightly worse than the reactive protocol (in the advertisement phase), and it performs slightly better than the proactive protocol (in the query phase).

6.4.3 Replay

When adversaries replay the same advertisements over and over again, they send outdated information in part of the network. This will lead to longer discovery times

and more false positives and false negatives. Adversaries spend little effort to replay advertisements, and the impact is getting larger when there are more adversaries. In the worst case, information in the entire network is outdated. Similarly as in the case of dropping advertisements, the reactive protocol is influenced the least, since no advertisements are broadcasted. If advertisements are not updated, `Ahoy` nodes do not propagate them, which will limit the impact of this attack significantly. If the proactive protocol also applies this principle, it has the same resilience as `Ahoy` in this respect.

When queries and replies are replayed, the network can be jammed. This we discuss further in Section 6.5.

One expensive, but very effective intrusion prevention solution for all replay attacks mentioned above is encryption. We discuss this further in Section 6.6.

6.5 Network Jamming

Jamming of the network is another attempt to make (requested) information unavailable for true users. The most severe attack in this respect is to flood packets into the network. The aim of the attack is to generate (partially) congested networks or nodes. Adversaries do not focus on the content of the exchanged packets, but just on the sheer amount of packets. Flooding packets can be especially harmful for ad-hoc networks, since most of the devices in such networks are battery-supplied. Adversaries can send massive amounts of messages with high frequency. As a result, batteries are exhausted and/or large portions of the limited bandwidth are occupied. Especially, we consider that a replaying attack has a similar effect as flooding. In the remainder of this section, we analyze the impact from flooding of different types of packets.

6.5.1 Flooding advertisement packets

In `Ahoy`, an advertisement is generated whenever there is a change in the outgoing filter of the node. This change can originate from the node itself or from the incoming filters of its neighboring nodes. If adversaries flood unchanged advertisements into the network (like replaying), they will not be propagated, since no changes

are detected. The only exhausted nodes will be the adversaries themselves. If adversaries flood different advertisements every time, the nodes within d hop range update accordingly, which will lead to exhausted nodes within a d hops range.

As mentioned before, the reactive protocol does not propagate advertisement packets, whereas the proactive protocol propagate advertisements to all nodes within d hops. Unlike `Ahoy`, proactive protocols, in general, do not exchange packets based on their contents. Whatever contents are in the advertisements, nodes propagate them to every node within d hops, which creates much more extra traffic than in `Ahoy`. However, proactive protocols can be adjusted such that they apply the same principle as in `Ahoy`, and limit the number of extra packets accordingly.

However, a drawback of `Ahoy` is that it consumes a lot of effort to remove information layer by layer as discussed in Section 5.3.2, while a proactive protocol can simply remove it at once. This scenario occurs whenever some context information types are removed in the newly broadcasted advertisements by adversaries, or when adversaries stop working and nodes realize some context information types have been removed.

One potential problem is the generation of extra false positive queries. When adversaries advertise different contents every time or the same contents for a long time, there is a high probability that they advertise information that does not exist. This will result in many false positive queries, which add an extra burden on the traffic load to the nodes in that area. This problem occurs in both `Ahoy` and proactive protocols. Some rules can be applied to detect the attacks and minimize the impact to the network as addressed further in Section 6.6.4.

We summarize the impact of flooding advertisement packets on the three protocols in Table 6.8.

6.5.2 Flooding query packets

The number of nodes that will be exhausted due to the flooding of queries depends on the type of protocol. Some nodes within d hops will be affected in `Ahoy`, no nodes are affected in the proactive protocol, and all nodes within d hops are affected in the reactive protocol. Especially, adversaries themselves are the first ones to be exhausted. In `Ahoy`, it is more difficult to harm many nodes by flooding query

Table 6.8: The vulnerability of three protocols when adversaries flood advertisement packets.

	Ahoy	**Proactive**	**Reactive**
Packet propagation range	all nodes within d hops distance when contents are different, no propagation when contents are the same	all nodes within d hops distance	none
Consequence	Partially influenced with high probability of false positives, no influence when replaying the same packets, lot of traffic to remove information	Nodes within d hops are affected, high probability of false positives	none

packets than by flooding advertisement packets. Most of the time, the queries can not be forwarded further if there is no match found. Therefore, the effects are minor for `Ahoy`, while the effects in the reactive protocol can be quite severe. In this respect, the reactive protocol has as clear disadvantage that it does not announce context information in advance.

One special case of flooding is replaying queries. If the adversaries simply replay the entire query packet with high frequency, the packets are dropped at the next hop. If these packets have the same QID, these redundant packets may be dropped, dependent on the applied rules. If adversaries generate queries with different QIDs, the same context information will be queried over and over again. The path to the queried information will then be heavily loaded. In this case, the proactive protocol performs the best with only one path congested. Depending on the query mechanism (sequential or parallel), one or several paths may be congested in `Ahoy`. These effects are much smaller than in reactive protocols. For these protocols, the whole d hops region might be jammed.

One effective solution to minimize the influence of query flooding is to restrict the query rate, which we discuss further in Section 6.6.4.

6.5.3 Flooding reply packets.

Reply packets unicast through the same path as the query packets, but in the opposite direction. Flooding replies will overload the nodes along the path, including the adversaries themselves. The effects are the same for all three protocols. The query node checks the QID in the reply packet. If there is already a connection established for queried information with the same QID, the remaining, redundant, packets are dropped. In case the QIDs are different and most likely not the ones from the query originator, the adversary is detected. However, before detection, it is still possible that a large number of packets have been sent between adversary and the targeted nodes.

To utilize an unique reply identification, RID, can help to avoid flooding same replies from a node. We can apply a rule to restrict per RID per reply per node. If a node detect few replies with the same RID from one node, it can detect that node as an adversary. Further, a rule to restrict the number of replies per QID, and per destination, could help to reduce the effects of reply flooding, but not solve the problem completely. If a couple of adversaries work together, for example, acting as a query node and a reply node, they can flood the network with queries and replies, while nodes along the path will not easily detect the attack. These nodes will therefore be quickly exhausted.

6.5.4 Summary

Flooding packets at a high rate will lead to extra traffic load. As a result, more bandwidth is occupied by malicious traffic, and targeted nodes may be exhausted.

In the advertisement phase, the reactive protocol will not be affected by flooding. `Ahoy` and the proactive protocol perform more or less the same. The exact performance of `Ahoy` and the proactive protocol depends on certain applied rules and on the contents of replayed messages. In the query phase, in `Ahoy`, only several paths can get congested due to flooding of queries. This is slightly more than in the proactive protocol, whereas in the reactive protocol, (part of) the whole network can be jammed. In the reply phase, all three protocols perform the same.

These results can also be viewed in terms of advertisement and query rates, such as was done in Chapter 4. There exist certain ranges of advertisement and query

flooding frequencies for which each protocol performs "best" (compared to the other protocols). When the query flooding rates are very high (compared to the flooding of advertisements), proactive protocols will be the least affected by flooding attacks. Like in Chapter 4, we expect that there is a large range of flooding frequencies (in which advertisement and query flooding rates are comparable) for which `Ahoy` will perform the best.

For all three protocols, there is no effective solution to protect against flooding attacks in advance. We can only minimize their effects by applying some rules, which are discussed in more detail in Section 6.6.4.

6.6 Countermeasures

`Ahoy` is designed for context-aware networks. It thus intends to publish available context information types, so that nodes are aware of existing context sources and know where to find requested information. To understand the broadcasted context information types from other nodes, some meta-information should be shared by all entities in `Ahoy`. The following list summarize this meta-information and how it is to be shared:

- the width and depth of ABFs: can be deducted from advertisement packets;
- hash functions in use: can be standardized and the number of hash functions can be added into the advertisement packet;
- context information type: can be standardized.

However, when this meta-information is shared or standardized, it also makes our network more vulnerable to attacks.

Most of the attacks are difficult to detect and to prevent, as we have discussed in detail in Section 6.3 through Section 6.5. But it is possible to avoid or minimize the effects of some attacks, e.g., by using text strings, applying unique QID and RID, restricting query rates, and using encryption. In this section, we focus on encryption strategies, the Michael algorithm (the message integrity code), authentication algorithms, and propose rule management to reduce the vulnerability of `Ahoy`.

6.6.1 Encryption

Encryption is the act of converting data from an understandable form to a non-understandable one in such a way that it can be converted back with no loss of information. Encryption is a good way to secure the contents of packets against undesired usage. It can be applied to particular fields of the packet, such as the ABF, QID, and RID, or to the entire packet. The encryption algorithm to be applied depends on the security requirements, and the available resources.

Based on the used key management scheme, modern ciphers can be categorized into several classes. Symmetric and asymmetric key cryptography are the two major ones. They are especially useful in ad-hoc networks [4]. Symmetric key algorithms use the same key for encryption and decryption. This requires the pre-distribution of keys to the communicating nodes. A drawback is that there are no guarantees for a reliable and secure way of key management. The advantage of symmetric key algorithms is that they are fast, because they utilize simple operations. Asymmetric key algorithms do not require an initial exchange of keys. A public key is used to encrypt the messages, while the receiver uses a private key to decrypt it. However, asymmetric key cryptography requires more complicated operations, such as modular exponentiation. For instance, the typical asymmetric key cryptography RSA performs 1000 times slower than the typical symmetric key cryptography Data Encryption Standard (DES) [19]. There is also a hybrid cryptography system, which uses asymmetric algorithms to distribute symmetric-keys at the start of a session [17].

Generally, expensive cryptography systems, are not preferred in `Ahoy` networks, because it may overload the nodes and makes the discovery slower. However, when the system requires very high security protection, a more sophisticated encryption algorithm should be used to protect the content of exchanged packets. The choice of symmetric, asymmetric, or hybrid encryption algorithms depends on such considerations.

The more simple symmetric key algorithms, for example, can be used in a conference environment. All the devices should be independently authenticated to join the network and pre-configured with symmetric keys. When a network has high security requirements, such as for military purposes, a hybrid approach can be used.

All devices should be authenticated to join the network, but in this case, an asymmetric algorithm is used to distribute a symmetric key when a session is initiated. During the session, this symmetric key is used to encrypt necessary information. If the session lasts long, the keys can also be updated with a certain frequency using an asymmetric algorithm. Due to the heavy computational costs, we suggest that asymmetric algorithms should not be used for the entire session.

6.6.2 Michael: Message Integrity Code

In protecting data integrity, especially against modification attacks, one option is to use Michael [23] to protect data corruption. Michael is a message integrity code (MIC) for temporal key integrity protocol (TKIP) for IEEE 802.11i draft [41]. It has specially been designed for preventing bit-flip attacks and a whole class of header modification. It uses a 64-bit Michael key to hash an arbitrarily long message. The result is a 64-bit Michael value, which is attached and sent together with the original packet.

With Michael, nodes can detect modification attacks, but at the price of larger packet size (including the Michael value). Besides, the nodes in the network also need to know which Michael key is in use. If adversaries know the Michael key, they can generate the Michael value after they modify a packet. Similar as in Section 6.6.1, this comes to the question of key distribution [10].

6.6.3 Authentication algorithms

Although `Ahoy` is a discovery protocol that tends to share available resources as much as possible and does not prefer authentication, it is still a good option to apply authentication algorithms, if to protect data within a group of users is a security goal. In reality, this also makes sense. If we assume that context information in a network should only be shared by a specific group of devices, it is reasonable to authenticate those devices before they join the group.

Typical authentication algorithms include message authentication code (MAC)

[57], username-password authentication, etc. Those algorithms have different complexity and security levels, and can be applied based on different security requirements. For example, in a normal conference ad-hoc network, simple username-password authentication scheme is probably sufficient.

After applying authentication algorithms, a pre-configuration file can be exchanged in a relatively safer environment. For example, such a pre-configuration file can include the meta-information, such as information about hash functions, and context information type standards. It can even include the keys for encryption or the Michael algorithm. In this way, any device that is authenticated to join the network has the necessary information to discover and share context information in the network. Adversaries can hardly modify any information in the network, if they lack the necessary information from the pre-configuration file.

6.6.4 Rule management

Encryption, Michael, and authentication are general approaches to enhance the security, which can be applied to any protocol. One specific countermeasure for `Ahoy` is rule management. As we introduced briefly in the previous sections, applying specific rules can avoid or reduce the impact from attacks. We summarize and address these polices as follows.

ABFs consistency check.

In the advertisement phase, adversaries can be detected by checking the consistency of their ABFs. What to do after detection, depends on the application requirements. For example, the node can broadcast the address of the adversary to the entire network, or to remove the adversary as its neighbor. Broadcasting the address of the adversary can lead to other problems that are related to trust. How reliable is the detection of an adversary? Who can be trusted? This is another complicated issue, which we will not cover in this thesis.

The following three rules can be used to check the consistency of ABFs.

- **Number of zeros in ABFs.** When all bits in all layers of the ABF are 0, no information is hosted by or can be accessed from this node. This scenario is

very rare. There is an exceptional case of a new node that joins the network, which does not have any context information. However, it is very unlikely that the neighboring nodes also do not have context information. After a while, there should therefore be at least some bits set to 1 in the lower layers of the ABF. If we spot a node with only 0s' in its ABF, and if it keeps this status for some time, we can assume that the node is an adversary. A rule to check whether all layers are 0 can be applied to detect probable adversaries.

- **Number of ones in ABFs.** When many bits are set to 1, there is a relatively high probability of false positives. In `Ahoy`, we avoid large number of false positives, by setting the ABF parameters w (width of the filter), b (the number of hash functions in use) accordingly, based on the number of context information in reach (related to d (depth of the filter), and x (the number of context information that can be reached in certain hops away)), as discussed in Chapter 4.

  The exact value of x can not be known in advance. However, by studying historical data or using other prediction algorithms, we can estimate it roughly. The results of Chapter 4 also show that when the exact number of context information types in the network is unknown or estimated, it might be even better to set w at a slightly larger value than the optimal setting for the estimated number of types. It will then not generate much more traffic, but it can accommodate more context information, if necessary, in the future. If w, b, and x are known, we can obtain the distribution of the number of bits that are set to 1, which we do in Appendix B. We can thus calculate the probability that the number of 1s' is larger or smaller than a specific number given that the node is not an adversary. This probability is very small when a lot of 1s' or little 1s' are set. In that case it is more likely that the node is an adversary.

  With this information, users of `Ahoy` should decide how to deal with the suspected adversaries. They can warn other neighbors by announcing this probability to other nodes in the network. They can also remove the suspected adversary as their neighbor. However, there is a risk of removing a normal node. This risk is higher when many nodes in a high density network are evaluated, although one might argue that the removal of one neighbor in such

a case does not really matter. Also, this rule can only be effective if at least some adversaries only add or remove 1s' to the filter. If adversaries only add (or remove) as many 1s' as they remove (or add), i.e., if they modify the filter without changing the number of bits set, they will have the same distribution of 1s' as normal nodes. In that case, this rule will not be effective.

- **Duplication of the information in the lower layer.** We duplicate local information in every lower layer of the ABF to reduce unnecessary transmissions when one context information is added (refer to Chapter 3.3.3). Therefore, the bits that are set to 1 in the higher layers should also be set to 1 in all lower layers. If this is not the case, the node can be an adversary or a malfunctioning node that cannot perform the duplication. Since the duplication is a rather simple operation, the probability that a node cannot perform this operation correctly is rather small. Nodes that detect such a mis-behavior node can determine that it is an adversary with great probability.

Restricted update rates

To avoid that adversaries constantly modify advertisements or flooding advertisements into the network, we can require nodes to wait for a small amount of time before it can send out an advertisement.

The timer, *advertisement-min-time*, should be set to a small value, for instance, 1 second. The process of updating generally spreads very quickly within the d-hop range. However, the updates should also be exchanged back and forth between neighbors for several times when context information is removed from the filters (an "advertisement loop") due to adding context information has been solved by duplicating information in the lower layer before broadcasting; see Chapter 3 for details. During the short waiting period, the node can collect the updates from neighbors and process them at once. This delay helps to reduce some traffic that is generated by too frequent updates.

Nodes should follow the following three actions when they send out an advertisement:

- **Action 1:** Whenever an advertisement is sent, the timer *advertisement-min-time* is set and starts to count down.

- **Action 2:** When a new ABF is computed, the timer *advertisement-min-time* is checked. If *advertisement-min-time* = 0, the new ABF can be sent out, and the timer *advertisement-min-time* is set again. If not, the new ABF is discarded.

- **Action 3:** Every time when the timer *advertisement-min-time* counts to 0, a new ABF is generated. The new ABF is sent out only when it is different from the last broadcasted ABF.

This rule may slow down the updating process slightly and can influence the discovery efficiency, especially in highly mobile environments. When changes in context information are not updated on time, more false positives and false negatives may appear. It depends on the setting of *advertisement-min-time*. A large value of *advertisement-min-time* may have a negative impact on the discovery efficiency. However, our purpose to set a timer is also to enable the nodes to complete the update process. We therefore suggest that the *advertisement-min-time* should depend on the time that it takes to complete the update process, which in general will be a short time.

Restrict the query and reply rate

Similar to the previous rule, we can restrict the query and reply rate to minimize the impact of the flooding by queries and replies from adversaries. We can use *query-min-time* and *reply-min-time* to count the timer. The following actions should be followed:

- **Action 1:** Whenever a query/reply is sent, the timer *query-min-time/reply-min-time* is set and starts to count down.

- **Action 2:** When a new query/reply is generated, the timer *query-min-time/ reply-min-time* is checked. If it is 0, the new generated query/reply can be sent out, and the timer is set *query-min-time/reply-min-time* again. If not, the new query/reply is hold and stored.

- **Action 3:** Every time when the timer *query-min-time/reply-min-time* counts to 0, nodes check the queued queries/replies. The first one in the queue will be sent out and the timer is set again.

This cannot eliminate the threats, but can reduce the danger of jamming the (entire) network. Both timers should not be set to a large value. Otherwise, it might lead to a long queue of queries/replies waiting for being sent out. This might seriously delay the discovery process. However, the timers should also be set large enough to minimize the effect from flooding attacks. The decision should be made based on the detailed network scenarios.

Meanwhile, this rule, as well as the previous rule, can lead to slightly larger discovery times and lower discovery efficiencies, especially in high mobility and high discovery frequency environments. However, we still recommend this rule in general, considering the positive contributions compared to the small drawbacks.

Withdraw packets reusing the same advertisement/query/reply ID.

To avoid replaying and flooding attacks, every advertisement, query, and reply message should have an unique identification. Especially, for each reply it should be clear to which destination it is sent, to which query it responds and from which context source it is sent. Nodes should drop a packet with the same ID as the previous packet. We have used GID to identify advertisements, QID to identify queries. We also suggested to use RID to identify replies in this Chapter.

In Section 3.4, we have defined the rule when a query packet is received that a query with the same QID and the same originator is discarded. Similarly, we apply the following rule when a advertisement packet is received:

- **Step 1:** Whenever one advertisement packet is received, the node checks the GID and the sender's address of the advertisement.
- **Step 2:** If the same sender has sent a packet with the same ID, the packet is discarded. Otherwise, the corresponding steps will be taken based on different packet types.

The following steps are performed when a reply is received:

- **Step 1:** Whenever one reply packet is received, the RID, QID, and the sender's address is checked.
- **Step 2:** If the same sender has send a packet with the same RID, the packet is discarded.

- **Step 3:** If the same sender of reply has a reply with the same QID, but with the different RID, the packet is discarded.

- **Step 4:** Otherwise, the reply is recorded by the node and forwarded.

This rule does not have any side effect, but can well avoid adversaries replaying the same packets to the network.

The hop-count of a query should never be larger than maximum hop counter d.

To minimize the impact of a modification of the hop counter of the query packet, the hop-count of a query should always be less than or equal to the maximum hop counter d.

The following steps should be taken to perform this rule:

- **Step 1:** Whenever a query is scheduled to be sent out, nodes check the current value of *hop-count*.

- **Step 2:** If the value is larger than d, it is set to d. Otherwise, no action is taken.

This rule can not eliminate the threats, but can avoid that adversaries discover information in the network more than d hops away. If there is only one adversary, the infected query can at most travel $2d$ hops away from the query originator, when the adversary is the dth hop neighbor of the originator and changed the value of *hop-count* from 0 to d. This happens when the adversary is located d hops away from the query originator. If multiple adversaries coordinate with each other and keep on modifying the *hop-count* to d, the query can be forwarded throughout the entire network in the worst case.

Since the rule can help to reduce the effects from this type of attack, and since it is not difficult to implement, we recommend to apply this rule.

6.7 Summary

In this chapter, we studied the vulnerability of `Ahoy`, the proactive and reactive protocols against the eavesdropping, modification, packet dropping, replaying, and flooding attacks towards exchanged discovery packets. In general, the three protocols are more or less equally vulnerable towards those attacks. For `Ahoy`, there is some benefit of using ABFs that contents are protected by hashing, while the exchanged contents of the other two protocols are directly readable by adversaries. On the other hand, a small modification in an ABF may lead to more changes in the contents than a similar modification in the original information. With respect to the extra traffic generated by attacks, `Ahoy` has the advantages of aggregating information from neighbors into one ABF and selective queries, but also has the disadvantage of "advertisement loops" to remove information.

The existing countermeasures, such as encryption, Michael, authentication can be equally applied to all three protocols. These countermeasures do not generate extra burdens for `Ahoy`. However, as usual, security comes with price. Applying countermeasures requires more computational power from nodes and generates extra traffic load.

Rules can be applied to improve the vulnerability. Some of the rules can be applied in all three protocols, such as to restrict update rates, to withdraw duplicated packets, and to restrict query range. But some of them which are based on the characteristics of ABFs, e.g., to check the consistency of ABFs, are only valid for `Ahoy`, and not for the other two discovery protocols.

Chapter 7

Proof-of-Concept Implementation

In the previous chapters, we have proposed and designed a novel context discovery protocol, `Ahoy`. We have evaluated the performance of `Ahoy` in terms of network traffic in both static and dynamic networks using analytical models and simulations. Now we are interested in how `Ahoy` performs in practice. In this chapter, we implement a prototype of `Ahoy`. We test the prototype in virtual networks to observe whether `Ahoy` performs as expected. We assess how much traffic `Ahoy` generates as a portion of the total traffic in the network, and we evaluate some of our design choices.

This chapter is written in close cooperation with Robbert Haarman. As part of his master assignment [31], Haarman has implemented `Ahoy` on top of UDP and IPV6 in Ruby [70]. Ruby is a simple and effective open source programming language, which can well support our needs to establish an interaction between `Ahoy` and the underlying protocol UDP and IPv6. Haarman has tested the prototype on UNIX-like (GNU/Linux) platforms. A virtual ad-hoc network consisting of virtual computers was set up to test the performance of the prototype, using User Mode Linux [79]. Each virtual machine runs a Unix-like platform (Debian GNU/Linux 4.0 (etch) [20]). The routing between virtual machines has been performed by using the OLSR [15] routing protocol.

In Section 7.1, we introduce the implementation choices for the prototype. In Section 7.2, we describe the message types and their formats. Section 7.3 gives an overview of the functional implementation of the prototype. Section 7.4 presents test results. Section 7.5 discusses the lessens learned from the prototype implementation.

The work described in this chapter has been published in [32].

7.1 Implementation Choices

We have defined the `Ahoy` protocol in detail in Chapter 3. There are three fundamental phases in `Ahoy`: context exchange, context query, and context maintenance. Context information types are hashed into attenuated Bloom filters (ABFs) and broadcasted to the neighbors. Nodes that have received ABFs from their neighbors, aggregate those ABFs, generate a new ABF, and broadcast it. In this way, one context information type can be propagated d hops away. Whenever a node looks for some context information, a query is generated. Nodes first check their local context information types. If there is no match, they check the stored ABFs from their neighbors. If a match is found, the node sends the query to the corresponding neighbor. In doing so, the query is propagated to the nodes which probably contain the requested information. The propagation of queries are stopped either when the requested information is found or when the query has been forwarded d hops away from the query originator. Whenever the requested context source is found, a reply is generated and sent back to the query originator. We implement the prototype based on this protocol definition.

In Chapter 3, we have discussed several design choices, which should be determined before implementation based on different network requirements and settings. Moreover, we encounter a couple of other options regarding the implementation, such as that of the underlying protocol. In this section, we first determine the choices for the prototype implementation in detail, according to our implementation requirements.

7.1.1 Context Information Type Format

In principle, the original context information types can be specified in any format. The general assumption is however that every type of context information should be uniquely known by a specific name, which all nodes are aware of. In the prototype, we describe context information types in a simple way by means of text strings. Each context information type is generated randomly so that it is unique over the network.

7.1.2 Context Duplication

In Section 3.3.3, we have discussed the advertisement loop, which occurs when information is added to the ABF, layer by layer. We have proposed two methods to solve this problem: applying an advertisement timer to delay frequent subsequent advertisements and context duplication. Context duplication can well avoid advertisement loops due to adding context information types. Applying a timer can only reduce the number of advertisement loops, depending on the settings of the timer. However, it provides the opportunity to observe the influence of advertisement loops. Because this is one of our goals for the prototype, we choose the first option to set an advertisement timer.

7.1.3 Query Format

Queries can be sent either in the original or BF format, as introduced in Section 3.4.2. Both formats have pros and cons. In the prototype, the original information is described in textual format which is not a space-consuming format. We choose to query in the original format in the prototype, so that we can ensure the discovered information is exactly the requested one. When we use BF format, there is a chance we obtain a false answer when multiple information is hashed into the same code. Here, we prefer to consume slightly more network traffic to query in textual format, instead of spending much effort to set a connection with wrong context sources.

7.1.4 Query Method

Queries can be sent in parallel to all possible matches at once or sequentially, one at a time, as was discussed in Section 3.4.2. Using parallel querying we are able to find all the possible context sources, while with the sequential querying only one source is found. The decision basically depends on the network scenario. In the prototype, we implement the parallel querying method.

7.1.5 Route Recording

We discussed three route recording options in Section 3.4.2. Here, we assume the availability of an underlying routing protocol. We use this protocol to deliver reply messages.

7.1.6 Means of Query Propagation

`Ahoy` requires a network that at least supports both broadcast and unicast. We broadcast advertisements to all neighbors and unicast replies to one specific neighbor. Queries, however, can be sent by broadcast or unicast. When a query is broadcasted, all neighbors receive it. Nodes need to spend extra effort to process the query, and to determine whether they have to take further actions. When a query is unicasted, only one specific neighbor receives it. However, using parallel querying, the query needs to be unicasted several times when multiple neighbors match the query.

The choice between broadcasting and unicasting queries depends on the network scenario. In wireless networks, a message is probably received by several neighbors, even if the massage was not for them. In this respect, broadcast is a more efficient way to propagate queries. Further, it depends on which query method is chosen. Unicast is suitable for the sequential query method, since messages are only sent to one node at a time. Broadcast is suitable for the parallel method, since more nodes need to act on the query at once. Finally, unicast provides the possibility to define different propagation scopes of query messages for different neighbors, depending on the number of hops in which a match is expected to be found.

In the implementation, we use the parameter *broadcast-queries* to set the method to propagate query messages. If it is set to *true*, queries are sent by broadcast. If it is set to *false*, queries are sent by unicast. By default, it is set to *false*.

7.1.7 Underlying Protocols Support

`Ahoy` has been designed independently from the underlying protocols. It is suitable to serve context discovery in different layers. It can reside on top of the transport layer, e.g., Transmission Control Protocol (TCP) or User Datagram Protocol (UDP),

on top of the network layer, e.g., Internet protocol (IP), or on top of the link layer, directly above Media Access Control (MAC) sub-layer. Because the operation of `Ahoy` does not necessarily need any support from the transport and network layers, it is most suitable to reside on top of MAC sub-layer, regarding the system complexity. However, the existing programming languages, e.g., Ruby, C, and Java, are lacking ready interfaces to MAC layer, but they have interfaces to TCP/UDP. For ease of implementation, we choose to reside `Ahoy` on top of UDP. UDP is a very simple protocol which only has 8 bytes of header. In `Ahoy`, advertisements and keep-alive messages need to be sent to all neighbors for update and maintenance. The most effective way to do it is via broadcast. Moreover, queries need to be sent to specific neighbors to perform directional querying. This can be done via unicast or broadcast (see Section 7.1.6). Since UDP supports both broadcast and unicast, it is an ideal platform to build our prototype on.

Further, UDP can be layered on top of both IPv4 and IPv6. With the explosive growth of the internet, IPv4 addresses are expected to get exhausted in the near future. IPv6 is the successor of IPv4, and is introduced to resolve the risk of address exhaustion. Therefore, we use IPv6 in the `Ahoy` prototype.

7.2 Message Type and Message Format

There are five types of messages in `Ahoy`: advertisements, queries, replies, keep-alives, and update requests, as introduced in Section 3.2. In the following part of this section, we are going to introduce their types and formats in detail [31].

7.2.1 Address

Before starting introducing the messages, we first define the address field, which is used in several types of messages, i.e., queries and replies. The address field can be defined as follows.

This field starts with an 8-bit *size* sub-field to indicate the size of the address field in bytes. In total, the size of the address field is 20 bytes if we use IPv6. As a result, it is set to 20. 8 bits are used to specify the type of the address, IPv6 or IPv4. For IPv6, it is set to 1. The field *port* declares the port number used by the

context information, if necessary. It is a 16-bit unsigned integer. The last 128 bits are used to store the IP addresses. Figure 7.1 shows the packet format of an IPv6 address.

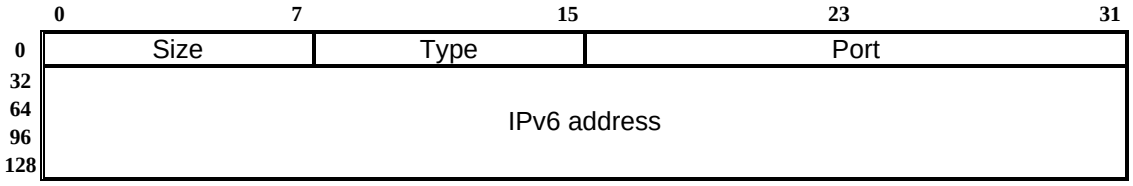


Figure 7.1: Packet format of an IPv6 address.

7.2.2 Advertisement

The advertisement message format is shown in Figure 7.2. The packet contains 8 bits of message *type*. For advertisement packets, the default type is 1. *GID* is a 32-bit unsigned integer to indicate the freshness of the information, see Section 3.5. We use 8 bits and 16 bits to store the information for ABFs' *depth* and *width*, respectively. As our basic assumption for `Ahoy`, all the nodes over the entire network should agree on the same *depth* and *width* for ABFs in use. By default, the depth is set to 4 and the width is set to 128 bits. The remaining part of the advertisement contains the broadcasted ABF. The size of the filter depends on the parameter depth (d) and width (w). It can be calculated by multiplying d and w.

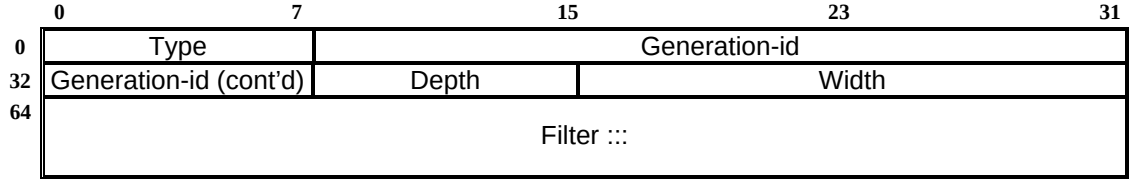


Figure 7.2: Advertisement packet format.

7.2.3 Query

Figure 7.3 shows the message format of a query. The 8-bit message *type* is set to 2 for query messages. We use a 32-bit unsigned integer, named *query-id*, to identify a query. 8 bits are reserved to indicate *time-to-live*, which represents the maximum number of hops that the query can still be propagated. The field *name-length* indicates the number of bytes needed to store the original context information, followed with a field *name* to store the entire query. Finally, the address of the query originator is included. According to Section 7.2.1, we need 20 bytes.

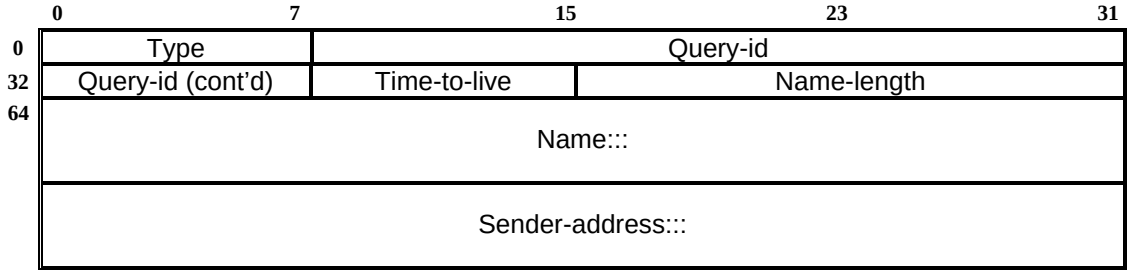


Figure 7.3: Query packet format.

7.2.4 Reply

The reply message format can be depicted in Figure 7.4. The 8-bit message *type* is set to 3 for reply messages. A reply message contains the 32-bit *query-id* it responds to. As introduced in Section 7.2.1, 20 bytes is used for the address of the replying node.

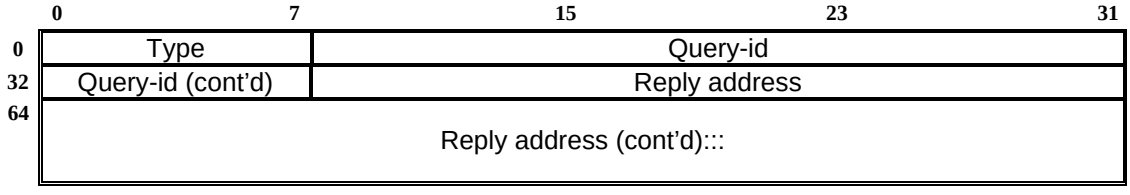


Figure 7.4: Reply packet format.

7.2.5 Keep-alive

The keep-alive message format is defined as in Figure 7.5. There are only two fields defined in the message: *type* and GID. The type for keep-alive messages is 4. GID is a 32-bit unsigned integer to indicate the freshness of the advertisement.

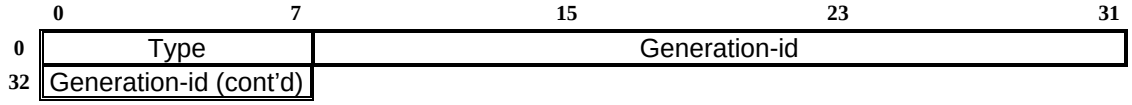


Figure 7.5: Keep-alive packet format.

7.2.6 Update request

The format is defined as follows in Figure 7.6. Only one field *type* is defined. For update request, it is set to 5.

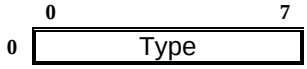


Figure 7.6: Update request packet format.

7.3 Functional Description

7.3.1 Event and State Variables

In this section, `Ahoy` is described in events and actions in response to these events. Four types of events are defined [31]:

- **User actions:** users can announce or revoke context type information they have, or query some information they are looking for.
- **Neighboring nodes:** neighbors can send advertisements, queries, replies, keep-alives, and update requests to the node arriving through a UDP port.

- **Timeouts:** several time outs have been defined for various purposes: minimum advertisement delay, keep-alive delay, query timeout, query cache cleanup timer, and service list cleanup timer.

- **Exceptions:** it can be caused by users sending a break to terminate the program, the operating systems sending a terminate signal, or out of memory, etc.

In the implementation, actions are triggered by different events. Meanwhile, five state variables (data structures) are maintained by `Ahoy` daemons [31]:

Query Cache The query cache is used for detecting (and subsequently discarding) duplicate queries. For each query that is received, it contains the query ID, the source address of the query, and a time stamp. The query cache is cleaned up at regular intervals by purging old entries. This time interval is counted by a query cache cleanup timer.

Neighbor List The neighbor list contains information about the direct (currently known) neighbors of the node running the `Ahoy` daemon. For every neighbor, it contains the neighbor's address, a time stamp, and the latest advertisement (GID and ABFs) received from that neighbor.

Local Services The local services list contains the context information types announced by users. For each context information type, it contains the name, the address, and a time stamp. The local services list is cleaned regularly by removing old entries. We apply a service list cleanup timer to determine the time for cleaning up.

Latest Advertisement The `Ahoy` daemon keeps the information sent out in the latest advertisement. When the neighbor list or the local services change, a new BF is computed, but these only need to be broadcast if they are different from the filters in the latest advertisement. Keeping a copy of the information sent out in the latest advertisement allows the daemon to decide whether it needs to send out a new advertisement or not.

Active Queries The daemon keeps a list of active queries, i.e., queries that have been initiated by users and that the daemon is currently awaiting responses

for. For each such query, the list contains the query id and the socket on which the user program is listening for responses.

The following part of the section, we address the functional flow in detail [31].

7.3.2 Initialization

After configuring the IP address and the port, a UDP socket is opened for sending and receiving `Ahoy` messages. A node broadcasts an update request through the socket and enters the IDLE state. It waits for the following events to occur to precede further actions [31].

- An advertisement is received from another `Ahoy` node (Section 7.3.3).
- A query is received from another `Ahoy` node (Section 7.3.4).
- A response is received from another `Ahoy` node (Section 7.3.5).
- A keep-alive message is received from another `Ahoy` node (Section 7.3.6).
- An update-request message is received from another `Ahoy` node (Section 7.3.7).
- An advertisement is received from a user (Section 7.3.8).
- A revocation is received from a user (Section 7.3.9).
- A query is received from a user (Section 7.3.10).
- The keep-alive timer expires (Section 7.3.11).
- The advertisement timer expires (Section 7.3.12).
- The query cache cleanup timer expires (Section 7.3.13).
- The service list cleanup timer expires (Section 7.3.14).
- A query timer expires (Section7.3.15).
- A signal is received from the operating system, causing the daemon to clean up and exit (Section 7.3.16).

Nodes handle these events as described below. When a received message does not fit in one of the above categories, we emit a warning and discard the message.

7.3.3 `Ahoy` Advertisements

When an advertisement is received from another node, the following steps are taken:

- The current time, the content of the advertisement, and the IP address of the neighbor from which it was received are recorded in the Neighbor List. The new information overwrites any previous entry for the same address.
- A new ABF is computed based on the new information and a new advertisement with this ABF is created.
- If the new advertisement differs from the latest advertisement, it is broadcasted to all neighbors and becomes the new latest advertisement.

7.3.4 `Ahoy` Queries

When a query is received from another node, the following actions are performed:

- The ID and source address of the query are looked up in the Query Cache. If a match is found (meaning the query has been seen before), only the time stamp in the cache is updated, and no further processing is performed, and the query is discarded.
- The current time, and the ID and the source address of the query are inserted into the Query Cache.
- The queried context information type is looked up in the Local Services. For any matching services, a reply message is sent to the query's source address.
- If the query's *time-to-live* is greater than 1, it is propagated. Propagation works as follows:
    - A new query message is created, with its ID, context information type, and source address equal to those of the received query, and a *time-to-live* of one lower than the received value.

- The Neighbor List is cleaned up by removing neighbors from which no advertisement or keep-alive message has been received in the last *neighbor-timeout* seconds. If any neighbors were removed, a new advertisement is computed based on the remaining service information. If this advertisement differs from the Latest Advertisement, it is broadcast to neighboring nodes, becomes the new Latest Advertisement, and the keep-alive timer is reset.
- After the Neighbor List has been cleaned, a look up is performed against it, returning all neighbors who have matches for the service name in their Bloom filters, within a number of hops less than or equal to the time-to-live of the new query.
- The new query is sent to all these neighbors, if any. If the parameter *broadcast-queries* is false, the query is unicast to each individual address. Otherwise, a single broadcast message is sent if there are any knowledgeable neighbors. No query message is sent if the look up in the previous item did not return any matches.

7.3.5 `Ahoy` Responses

When a response is received from another node, it is processed as follows:

- The ID of the response is looked up in the Active Queries. If no match is found, no further processing is done.
- If a match is found, the address contained in the response is sent to the user program waiting for it.

7.3.6 Keep-Alive Messages

When a keep-alive message is received from another node, it takes the following actions:

- The GID is extracted from the keep-alive message.

- The node looks up the neighbor who sent the keep-alive message in the Neighbor List.
- If the neighbor is found in the list, and the GID recorded in the entry in the Neighbor List matches that of the message, the timestamp for the entry is updated.
- If the neighbor is not found in the list, or if the recorded GID does not match the one contained in the message, an update-request is sent to the neighbor.

7.3.7 Update-Request Messages

If an update-request message is received, the node sends its latest advertisement to the neighbor that sends the update-request.

7.3.8 User Advertisements

Users can announce context information types on `Ahoy` by specifying the name of the context information type, the IP address, and the port number. User advertisements are handled by taking the following steps:

- A user enters the context information type, the port number, and the IP address in the Local Services. If an entry with the same type and address already exists, the existing entry is overwritten by updating the time stamp.
- Based on the new information, a new ABF is computed.
- If the new ABF differs from those sent in the Latest Advertisement, a new advertisement with the updated ABF is sent out, and recorded as the new Latest Advertisement. The keep-alive timer is rest.

Note that, local services are automatically removed after *local-service-timeout* second. Users have to re-announce their services periodically to keep them available in the network.

7.3.9 User Revocations

Users can revoke advertisements by specifying the context information type, the IP address, and port number of the advertisement to be revoked. `Ahoy` performs the following steps to revoke advertisements:

- A user looks up the context information type and the address in the Local Services.
- If no entry is found, no further processing is done.
- If an entry is found, it is removed.
- When the last entry for the given context information type removes, the information type is no longer available. A new ABF is computed. If the new advertisement with the new ABF differs from the Latest Advertisement, it becomes the new Latest Advertisement. The new advertisement is distributed to all neighbors. The keep-alive timer is reset.

7.3.10 User Queries

Users can query context information type by performing the following steps:

- A user enters a context information type to query.
- An ID is generated for the query and recorded into the Active Query, along with information about the user program that responses are to be sent to.
- Generate a query message containing the ID, the context information type, the IP address, and a *time-to-live* field with a value equal to depth.
- Search the Neighbor List for neighbors that have matches for the service name in their Latest Advertisements.
- Send the query to any such neighbors, using uni-cast if *broadcast-queries* is false, using broadcast otherwise.
- Set a timeout of *query-timeout* seconds. When the timeout expires, the query information is removed from Active Queries.

7.3.11 The Keep-Alive Timer

The keep-alive timer is used to schedule the next keep-alive message. Whenever an advertisement or a keep-alive message is sent, the keep-alive timer is set to

$$\textit{keep-alive-time} + \left(\textit{keep-alive-time} \cdot (\textit{rand} - \frac{1}{2}) \cdot \frac{\textit{keep-alive-jitter}}{100\%} \right).$$

The parameter *rand* is a random value between 0 and 1, while *keep-alive-jitter* is used to vary keep-alive period with a small randomness to avoid collisions. By default, *keep-alive-time* is 15 second and *keep-alive-jitter* is 25%. When the timer is expires, a keep-alive message is broadcasted to all neighbors.

Finally, when the keep-alive timer of a neighbor expires, the Neighbor List is also cleaned up by removing neighbors from which no advertisement or keep-alive messages have been received in the last *neighbor-timeout* seconds. A new ABF is generated based on information available from the remaining neighbors and Local Services. If the computed advertisement differs from the Latest Advertisement, it becomes the new Latest Advertisement and is broadcasted to neighboring nodes. The keep-alive timer is reset [31].

7.3.12 The Advertisement Timer

The advertisement timer is used to prevent frequent updates as we introduced in Section 7.1.2. The advertisement timer is set whenever a new advertisement has been generated, but the last broadcasted advertisement was sent less than *advertisement-min-time* seconds ago. The new advertisement cannot be sent immediately, and thus is scheduled to be sent at a later time.

When the advertisement timer expires, an advertisement computed from the latest available information (including changes that occurred after the timer was set) is sent. The new advertisement then becomes the new Latest Advertisement. The keep-alive timer is reset.

7.3.13 The Query Cache Cleanup Timer

The query cache cleanup timer is used to periodically clean up the Query Cache. It is set to *query-cache-timeout* seconds. It is first set when `Ahoy` is started, and

reset each time it expires. Whenever it expires, the Query Cache is cleaned up by discarding from the cache any entries that are older than *query-cache-timeout* seconds.

7.3.14 The Service List Cleanup Timer

The service list clean-up timer is used to clean up the Local Services list. It is set to *local-service-timeout*/10 seconds when `Ahoy` starts, as well as each time it expires. When it expires, any entry which is older than *local-service-timeout* seconds in the list of Local Services is removed. Removing an entry causes a new advertisement is generated. If the advertisement differs from the Latest Advertisement, it is broadcasted to neighbors. The keep-alive timer is reset.

7.3.15 Query Timeouts

Query timeouts are used to count the life time of a query. If no requested context information type has been found, there is no response sent back to the query originator to announce that. Whenever a query is initiated by a user, a timeout of *query-timeout* seconds is set. When the timeout expires, the query is discarded. Any responses that come in after such time will be ignored.

7.3.16 Shutdown

When the `Ahoy` daemon exits, it deletes the local socket it created for communicating with user programs.

7.4 Testing and Results

7.4.1 Test Goals and Settings

Tests have been done to validate the protocol. The tests are set up to help us to pursue the following objectives:

- To validate whether the prototype works as expected and whether all the messages are transmitted at the right time in a right order.

- To analyze the traffic (bytes per second) generated by `Ahoy` as fraction of the total traffic (including UDP, IPv6, and Ethernet).

- To observe the effect of advertisement loop when context is **not** duplicated in lower layers of ABFs.

Due to the different purposes, the traffic analyzed in the prototype is different with the traffic analyzed in the analytical and simulation models in Chapter 4 and 5. First of all, we implement the keep-alive mechanism here. In the analytical and simulation models, keep-alive messages are not taken into account, due to the focus on the performance study of updates with constant frequencies. Secondly, in contrast to the analytical and simulation models, we did not implement context duplication in the prototype, because we wanted to observe the influence of the advertisement loop in real world tests. There are therefore more packets transmitted in the prototype than in the models due to advertisement loops. Thirdly, the `Ahoy` query traffic in the prototype includes the real queries, false positive queries, keep-alive messages, and reply messages. In the analytical and simulation models, we aim to minimize the false positive probabilities and count only false positive queries. With the optimal settings obtained from Chapter 4, false positive queries can be only a (very) small part of the total `Ahoy` query traffic in the prototype. Therefore, we cannot directly compare the traffic from the prototype with the results from the analytical and simulation models.

In the following tests, we use TCPDUMP [76] to collect the traffic data. The `Ahoy` parameters and their settings are listed in Table 7.1.

7.4.2 Test scenarios

Tests have been done for both a 5- and a 13-node scenario, respectively. For each scenario, three different network topologies are examined: a network with full connectivity, a regular grid structured network, and a network with dynamic connectivity.

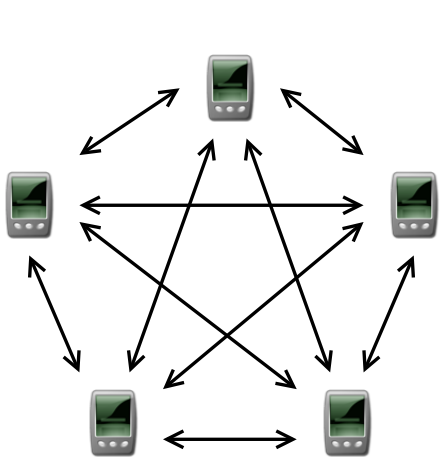
Table 7.1: Parameter Settings.

Parameter	**Value**	**Parameter**	**Value**
announcement-min-time	5 sec	*broadcast-queries*	true
depth	4	number of hash functions	3
local-address	/tmp/ahoy/socket	*port*	5000
keep-alive-time	15	*keep-alive-jitter*	25
local-service-timeout	300 sec	*query-timeout*	10 sec
query-cache-timeout	60 sec	width	128 bits

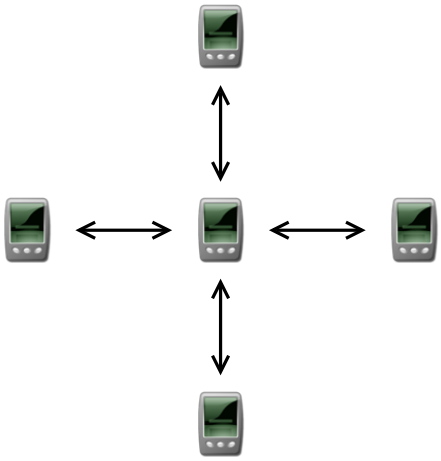
- Network with full connectivity: every node can connect to any other node in the network directly. Figure 7.7(a) shows an example of a fully connected five-node network. Please note that this figure does not depict the geographic topology of the network. As long as every node can reach the others directly, it fits the requirement of such a network. In the remainder of the section, we name this scenario as the "Full" scenario.

- Regular grid structured network: as described in Chapter 4. Figure 7.7(b) and Figure 7.7(c) present the abstract network topologies at the connection level for networks with 5 and 13 nodes, respectively. In the remainder of the section, we name this scenario as the "Grid" scenario.

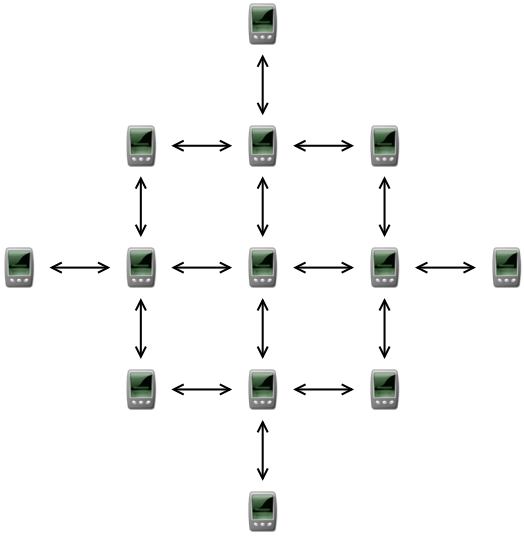
- Network with dynamic connectivity: nodes periodically and randomly change connections between each other. Every 30 seconds, the connections between nodes are randomly generated. The algorithm is as follows: all connections between each pair of nodes are initially enabled. Then, each node randomly decides whether or not to block communication to another node for the next 30 seconds. Each connection has 50% probability to be blocked. Since the network topology is essentially randomly determined every 30 seconds, the number of connections is not known in advance. There is no guarantee that all nodes are connected for each random topology. There is also no guarantee that all links are bi-directional. In the remainder of the section, we name this scenario as the "Dynamic" scenario.



(a) Full connectivity (5-node).



(b) Grid structure (5-node).



(c) Grid structure (13-node).

Figure 7.7: Examples of network structures [31].

For each topology of every scenario, three different sets of experiments have been performed, as follows.

- Experiment "Idle": to observe the performance of `Ahoy` in a stable state. In the experiments, `Ahoy` is idle and no context distribution or discovery is performed. Nodes only exchange keep-alive messages, update requests, and advertisements as requested. Note that update requests and advertisements are only sent when the network topology changes.
- Experiment "Advertise-revoke": to test the performance of `Ahoy` advertisements. In those experiments, one node advertises a context information type, and revokes it after 60 seconds, and advertises it again after 60 seconds, and so on.
- Experiment "Query": to evaluate the discovery ability of `Ahoy`. One node advertises a context information type, and another node is requested to query the context information type every 20 seconds.

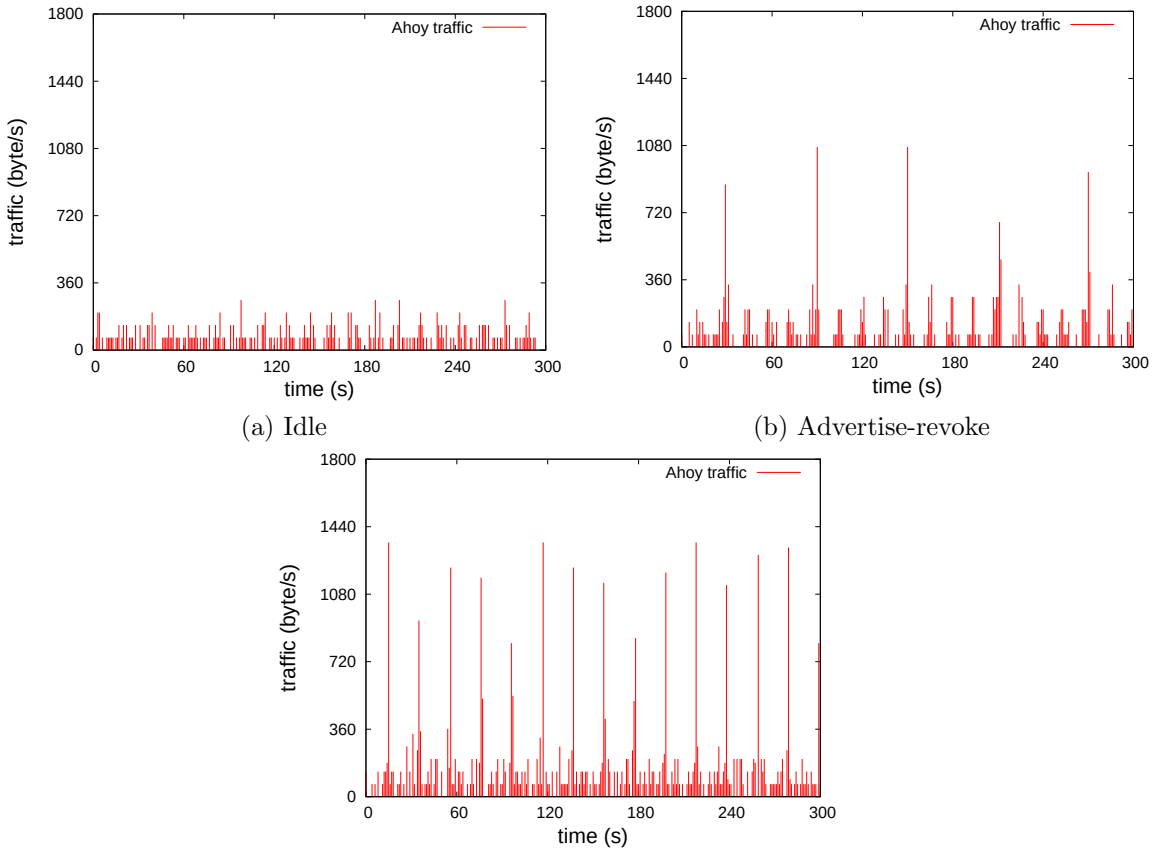
7.4.3 Test results

All the tests run for 300 seconds to collect enough data for calculating average network traffic (bytes per second). For each of the test setups, we collect and analyze the network traffic generated by `Ahoy` and the total network traffic (byte counts include UDP, IPv6, and Ethernet headers). Virtually all network traffic not generated by `Ahoy` is generated by OLSR (the routing protocol used), with a negligible portion of the traffic being ICMPv6 [16] control messages [31]. Figures are only plotted to visualize traffic generated by `Ahoy`. We present in detail the experiments done for 13-node networks. The experiments for 5-node networks present similar results to corresponding experiments in 13 nodes networks. Please refer to [31] for the detailed results of 5-node networks.

Full Connectivity

Figure 7.8 shows the test results for a 13-node network with full connectivity. Figure 7.8(a) shows the network traffic generated when `Ahoy` is idle. In this experiment,

there is no change in the network topology. Only a small amount of traffic is generated (60 bytes per second in average). Since there are only keep-alive messages generated in the network, there is no large peak generated in Figure 7.8(a). Figure 7.8(b) shows the case where information is periodically advertised and revoked. The figure show that every 60 seconds there is a peak of `Ahoy` traffic, which is generated due to broadcast ABFs for updating changes. Figure 7.8(c) shows the scenario where context discovery is performed. In the figure, there are very clear peaks of `Ahoy` traffic every 20 seconds, which correspond to the query messages.



(a) Idle

(b) Advertise-revoke

(c) Query

Figure 7.8: Traffic generated by a 13-node network with full connectivity [31].

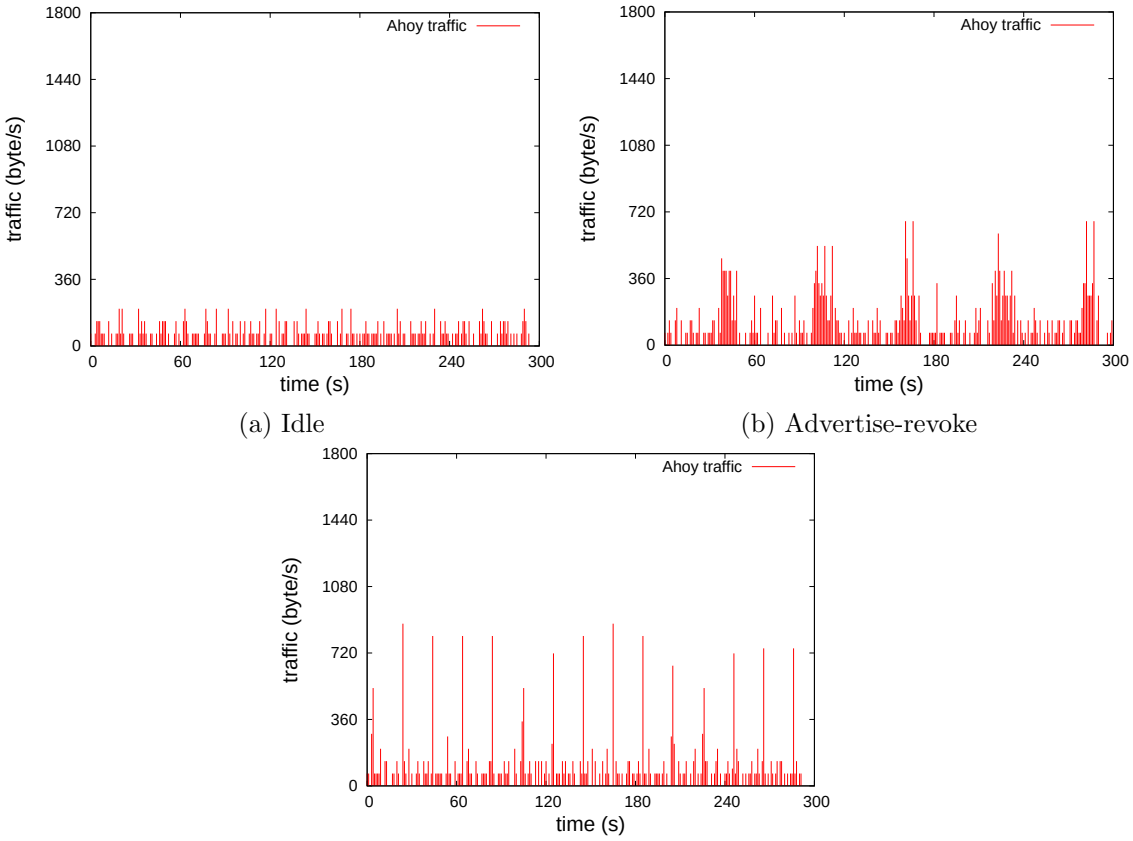
Grid structure

Figure 7.9 for grid structures share the same properties with the ones for full connectivity structures (Figure 7.8). When `Ahoy` is idle, only light-weighted keep-alive messages are transmitted in the network. When information is periodically advertised and revoked, the same peaks appear every 60 seconds in the figure. High peaks are generated every 20 seconds due the queries in the scenario of context discovery. Compared to Figure 7.8 from the "full" scenario, both scenarios have the similar pattern when `Ahoy` is idle. In Experiment "advertisement-revoke", more traffic is generated every 60 seconds. That is because more updates are needed in the "grid" scenario, due to "advertisement loops" generated when information is added or removed in the network. This effect is less visible in the "full" scenario, when every node can reach each other within one hop. On the other hand, more traffic is generated in the "grid" scenario than the "full" scenario, in Experiment "Query". Since we apply parallel query methods here, in the "full" scenario, all nodes are visited by the query, while only some of nodes are visited in the "grid" scenario.

Dynamic Connectivity

The network topologies change in every 30 seconds. When the connectivity between nodes changes, network traffic is generated to propagate the changes in the network.

Figure 7.10 demonstrates the network traffic generated by `Ahoy`. Figure 7.10(a) shows the case when `Ahoy` is idle. Unlike the static networks in the previous two scenarios, more peaks appear to reflect the change of topologies. Connectivity between nodes changes every 30 seconds. However, the changes take longer time to propagate to the nodes in range. Therefore, the peaks are not only concentrated on the points when changes happen, but spread further. Figure 7.10(b) presents the case when information is advertised and revoked periodically. It is clear that both the topology changes and queries contribute to the peaks in the figure. Similar effects are confirmed in Figure 7.10(c). Traffic generated by the change of topologies leads to more frequent and higher peaks than in the case of static networks.



(a) Idle

(b) Advertise-revoke

(c) Query

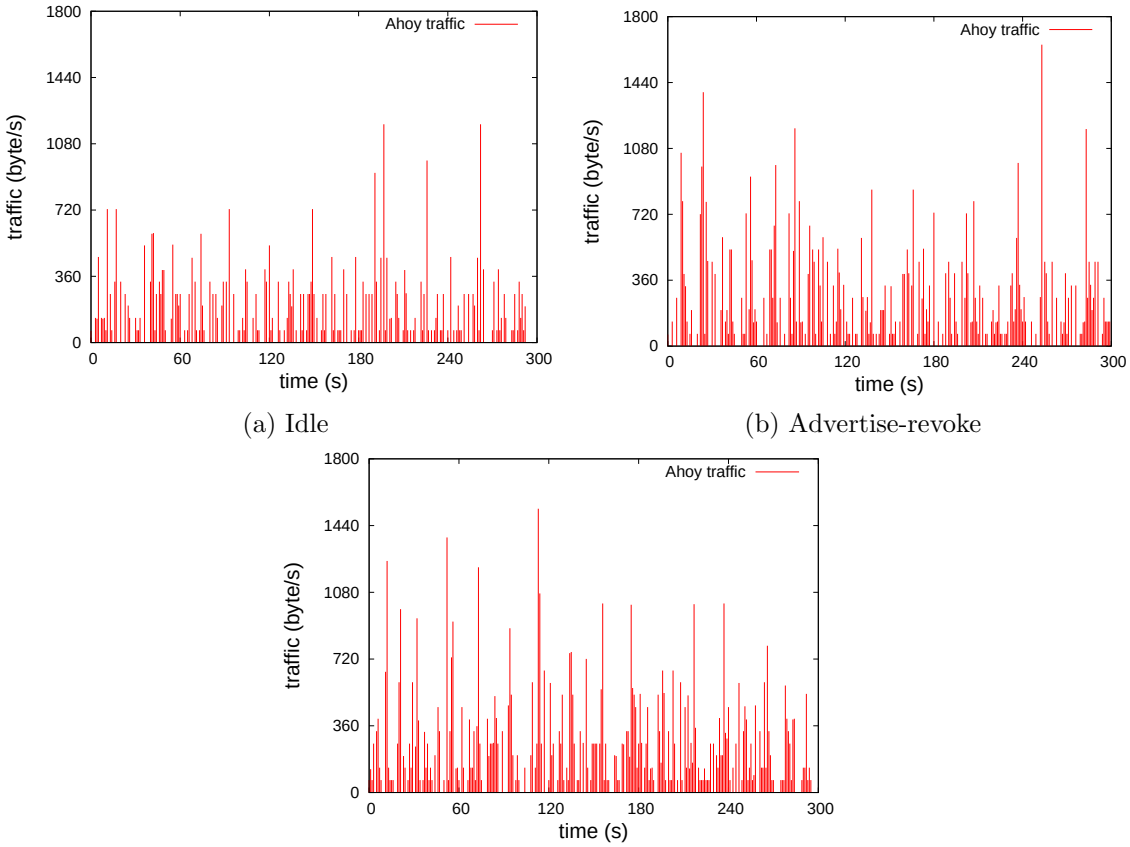
Figure 7.9: Traffic generated by a 13-node network with grid connectivity [31].

Summary

Figure 7.8, 7.9, and 7.10 demonstrate the expected behavior of `Ahoy`. The prototype performs as expected in all three experiments (idle, advertise-revoke, and query) of three cases (full, grid, and dynamic). Table 7.2 shows the average traffic generated by `Ahoy` (per node and the total `Ahoy` traffic) and the total traffic (including `Ahoy`, OLSR, and ICMPv6 traffic) in bytes per second for both 5-node and 13-node networks. We can make the following 3 groups of observation from the results:

1. **`Ahoy` performs consistently.**

    1.1 Nodes can discover and locate the requested information using `Ahoy`. In



(a) Idle

(b) Advertise-revoke

(c) Query

Figure 7.10: Traffic generated by a 13-node network with dynamic connectivity [31].

"Query" experiments, one node can successfully find an available context information type which is located in another node. This indicates that `Ahoy` properly advertises existing information types in the network and that queries are forwarded to the nodes which contain the requested information.

1.2 When there is no change in the availability and location of context information, and if the network topology does not change, `Ahoy` only produces keep-alive messages. In that case, the `Ahoy` traffic per node should remain the same, independently of node density. This has been shown in

Table 7.2: Test results (unit for network traffic: bytes per second).

		Idle			**Advertise-revoke**			**Query**		
		`Ahoy` Traffic		Total Traffic	`Ahoy` Traffic		Total Traffic	`Ahoy` Traffic		Total Traffic
		per node	total		per node	total		per node	total	
5 nodes	Full	5	23	402	10	50	412	10	52	437
	Grid	5	23	337	10	51	337	12	57	406
	Dynamic	7	36	510	10	49	480	14	70	512
13 nodes	Full	5	60	1809	6	81	1511	11	144	1756
	Grid	5	61	1436	9	117	1535	8	99	1560
	Dynamic	11	145	2938	16	204	2945	17	226	3017

Experiment "Idle". For both the "full connectivity" and "grid structure" scenarios, `Ahoy` traffic per node is 5 bytes per second. For both topologies, `Ahoy` traffic increases during the "Advertise-revoke" experiment. This result is also expected because in that experiment extra advertisements are needed to propagate updates through the network. For example, in the 5-node network with full connectivity, every node generates 10 bytes per second on average during the "Advertise-revoke" experiment.

1.3 More traffic is generated when the network is dynamic, because extra advertisements are sent due to changes in topology. For example, in the 13 nodes "Idle" scenario, 60 bytes per second and 61 bytes per second are generated in the "full connectivity" and "grid structure" topologies, respectively, while 145 bytes per second are generated in the "dynamic connectivity" network. Among those 145 bytes, 85 bytes are generated due to changing connectivity. When information is advertised and revoked periodically, 125 out of 204 bytes per second are generated due to updates. When querying is performed, 99 out of 226 bytes per second are generated due to updates and replies.

2. `Ahoy` **generates small amounts of traffic compared to the total traffic.** In Table 7.2, the "total traffic" refers to the sum of the traffic generated by `Ahoy`, OLSR, and ICMPv6. In general, we can observe that `Ahoy` generates a small portion of the total traffic. For example, among the above experiments,

the maximum fraction is 16.9% in Experiment "advertise-revoke" of the 5-node grid scenario, and the minimum fraction is 3.9% in Experiment "idle" of the 13-node dynamic scenario. In fact, with more nodes involved in the network, the `Ahoy` traffic generated per node, mostly remains the same or changes slightly, while OLSR traffic increases more strongly. Hence, the `Ahoy` traffic as fraction of the total traffic decreases when the number of nodes in the network increases. For example, in Experiment "idle" of the dynamic scenario, the fraction is 7.1% in a 5-node network, and it decreases to 3.9% in a 13-node network.

3. **Advertisement loops generate redundant traffic.** In Experiment "Advertise-revoke", for the full connectivity and grid structure, `Ahoy` generates more traffic per node in the 5 nodes networks than in 13 nodes networks. This is the result of the advertisement loop introduced in Section 3.3.3. To add information in ABFs, advertisement messages are propagated back and forward between neighbors multiple times, until the new information is added in all corresponding layers of the ABFs. A minimum delay *advertisement-min-time* is set between successive advertisements to merge multiple updates in one time slot of *advertisement-min-time*. This reduces the amount of network traffic to some extent, but cannot solve the problem completely, with the default setting of 5 seconds.

   Especially, the advertisement loops generate more traffic in a network with the grid structure topology than with the full connectivity topology. Because, all nodes are connected to each other in full connectivity networks, where all messages are received by all nodes. Updates due to the changes take several hops to reach all nodes in the grid network, while they take only one hop in the full connectivity networks. The more hops there are to reach each other, the more advertisement loops there are. Therefore, the difference is not so apparent in the 5-node grid structured networks, due to the limited depth of the network.

   This effect is less important in some dynamic connectivity scenarios. It can be even eliminated, because neighbors anyhow need to exchange ABFs with each other due to the frequent change of topologies.

7.5 Discussion

The prototype implementation shows us the possibility to implement Ahoy in the real world. The current protocol design of Ahoy, introduced in Chapter 3, does not need to be modified for the implementation purpose. However, couple of issues need to be further clarified to implements Ahoy as we discussed in Section 7.1. The decisions are dependent on the specific network scenarios, and can be made by the users.

Based on the experience of prototype implementation and tests, the following extensions could be beneficial [31]:

1. **Compatibility:** Currently, the `Ahoy` prototype is specifically implemented to work with UDP/IPv6: it uses UDP/IPv6 for sending and receiving messages; the addresses included in queries are IPv6 addresses; it sends responses to queries consist of an IPv6 address and a port number. `Ahoy` could be extended to support for other types of network, for example by adding other types of addresses, such as IPv4 addresses and Ethernet MAC addresses; or by defining completely different response types, such as URIs for services implemented on top of XML-RPC [82], or CORBA [9] identifiers.

2. **Self-configuration:** In the current version of `Ahoy` prototype, all nodes agree on the following parameter settings to the operation of `Ahoy` that must be identical for all nodes in the network, by default: the depth, d, and the width, w, of the ABFs, the number of hash functions being used, b. Based on our study in Chapter 4, optimal values for these parameters exist, but depend on the properties of the network and the services being offered. Currently, there is no way for nodes to automatically agree upon these values, and no way for nodes to change the values in response to changes on the network. It is necessary to enhance the function for `Ahoy` to allow `Ahoy` nodes automatically to decide on common values and react to changes in the network, for a mobile network.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Wireless technologies have developed rapidly in recent years. Due to increased bandwidth and more available applications and services, ad-hoc networks can be applied in various applications, from small scale personal environments to large scale remote, and open areas. In the future, ad-hoc networks will probably be used even more frequently, because they are easy to set up, economic, and more importantly, they are dynamic. On the other hand, ad-hoc networks have some limitations compared to fixed networks, such as relatively limited bandwidths, unstable wireless connections, and battery-powered devices. Context is any information about the characteristics of existing entities in the network, such as devices, applications, etc. When a node wants to look for some context information from the present network, those limitations pose challenges on the efficient discovery of context information.

In this thesis, we have presented a novel context discovery protocol `Ahoy` for ad-hoc networks. It uses space-efficient attenuated Bloom filters (ABFs) to represent context information types, and it supports an efficient directional context discovery. ABFs represent the availability of context information types within different numbers of hops. With ABFs, nodes can discover context information types with a small chance of false positive probabilities.

The **contributions** of this thesis can be summarized as follows.

- **A novel context discovery protocol `Ahoy` is proposed.** `Ahoy` has been proposed to perform context discovery for context-aware MANETs. Nodes can

have an overview about the available context information types within range, whenever they join an `Ahoy` network. Using that information, nodes can locate the relevant context information types fast and traffic-efficiently. Instead of storing the entire information, ABFs represent the availability of all essential information within d hops in just a few bytes. Queries are only forwarded to selected directions with a small chance of false positives.

- **An analytical model of `Ahoy` has been established.** Based on the protocol definition, we have developed an analytical model for `Ahoy`. The model is applied to two different network structures: simple grid networks and circle networks which represent fully distributed ad-hoc networks. The model calculates the overhead cost for advertisements and queries, during the discovery phase. It can be used to optimize the system parameter settings to minimize the network cost that is generated by `Ahoy`. It can also be used as a tool to evaluate the performance of `Ahoy` in comparison with purely proactive and reactive protocols. It is validated by simulations to be a very good tool to compute the proper size of the attenuated Bloom filters, for which the overhead network cost is minimized.

- **The performance of `Ahoy` in both static and dynamic networks has been evaluated.** Using the analytical model and the simulation model developed in [27], we have evaluated the performance of `Ahoy` in various networks. We have studied the overhead cost of `Ahoy`, including advertisements and false positive queries, in static networks, and compared them with the overhead cost generated by proactive and reactive protocols. Further, we have examined the number of updates generated by `Ahoy` in various scenarios in dynamic networks and compared again with proactive and reactive protocols.

- **The vulnerability of `Ahoy` has been studied.** We have analyzed the vulnerabilities of `Ahoy`. `Ahoy` has comparable vulnerability agains malicious attacks towards exchanged contents, compared with proactive and reactive protocols. The effects of various attacks have been studied in a qualitative way, and possible solutions have been discussed and proposed. Besides the inherent security weaknesses of ad-hoc networks, especially in wireless transmissions, some of the attacks to `Ahoy` itself can be detected and prevented by executing

the proposed rules. The existing countermeasures, e.g., encryptions, Michael [23], etc., can also be applied to `Ahoy` without generating extra burdens.

- **False positive probability for ABFs can be expressed exactly.** Until recently, an expression proposed by Bloom [6] was considered to be the exact false positive probability for Bloom filters, and it has been widely used in calculating this probability. In this thesis, we point out that this expression is only an approximation, and we derive the exact expression. The difference between the two expressions has been analyzed. From this, we show that the approximate expression, proposed by Bloom, is still a good approximation for high density networks. Due to its much simpler calculation, it is used in our analytical models for high density networks.

The main **conclusions** of the performance evaluation of `Ahoy` can be summarized as follows:

- `Ahoy` **generates significantly less (up to an order of magnitude) traffic load in a fully distributed ad-hoc network, compared to conventional approaches.** The use of ABFs significantly reduces the amount of traffic due to advertisements and queries. Compared to the proactive and reactive protocols, the performance of `Ahoy` depends on the ratio of query and advertisement rates, on the query range of nodes, and less so on the density of context information sources and node density. The main conclusion is that for practical situations, in a fully distributed ad-hoc network, `Ahoy` generates significantly less (up to an order of magnitude) overhead traffic load than advertising a full map of all available context types, or broadcasting queries when no advertisements are used. As such, it is a very promising compromise between the two extreme protocols, i.e., the proactive and reactive protocols.

- `Ahoy` **has good scalability and performs stable in dynamic environments.** We examined the performance of `Ahoy` for the following dynamic scenario's: node appearance, node disappearance, packet loss, and the movement of a node. From the analysis, we conclude that network load increases almost linearly with node density when nodes appear, disappear, or are temporarily unreachable. This proves that `Ahoy` is scalable in dynamic environments, and

that increased network densities do not lead to an explosion in network traffic. Instead, it results in a steady and slow almost linear increase in `Ahoy` traffic. Further studies have been performed for the case that one node is moving across the network. The simulation results show that the network load decreases slowly with increasing node speed. This is in line with our reality requirements. If a node is moving fast, it will not be an ideal context source provider. We do not want to waste network traffic to momentarily update the information of a fast moving node. Again, we find an almost linear relation between network load and network density, which proves that `Ahoy` is also scalable in more complex mobile environments.

These conclusions answer the research questions that were posed in Section 1.4. It shows that `Ahoy` is a very promising discovery protocol for ad-hoc networks. `Ahoy` generates relatively little traffic to announce and discover context type information in the network, and it uses simple operations in doing so. It behaves well; not only in small-scale networks, but also in big and dense networks; not only in a static network, but also in a dynamic environment.

8.2 Future Work

`Ahoy` is already a space- and traffic-efficient protocol, but further improvements are possible. In the experiments, we observed that there is much less traffic generated when context information is added than when it is removed. In particular, most traffic is generated when context information gets out of reach of some nodes, but still can be reached by other nodes. Reducing broadcast traffic when context sources are removed, therefore, is an important topic for further study. A radical solution, which will affect all traffic, is just to restrict the input information based on the quality of context. Only high-quality information should be added into the filters.

As mentioned in Chapter 5, `Ahoy` can be integrated with underlying routing protocols to limit the flooding of route requests. Generally, route requests are flooded through the entire network in DSR and AODV [71]. When `Ahoy` is applied, a node would query for context information and get a reply including an address and explicit (DSR) or implicit (AODV) routing information.

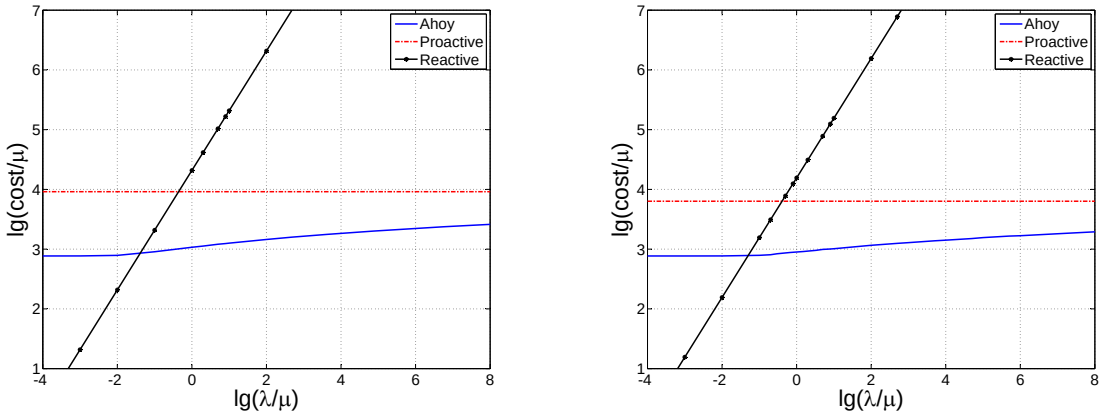
We can also improve `Ahoy` by making it more "context aware" and "self-adjusting". It should adapt its own behavior to offer optimal network traffic based on the current network situation. The performance will be improved if `Ahoy` can automatically adjust for all nodes in the network the Bloom filters' parameters, like the depth, the width, the number of hash functions, and the keep-alive period, can automatically choose between parallel query method or sequential query method, and can automatically apply different actions based on different security requirements. The protocol design would have to be extended to a system that is based on zones where nodes are grouped according to certain characteristics, such as their geographic positions. The extension of the protocol should then address the important issue of the transition of discovery processes between zones.

Finally, as we discussed in Chapter 6, `Ahoy` can be further expanded as a policy-based discovery protocol to enhance the network security.

With this future work, it is possible to develop an adaptive, slim, context-aware, and secure discovery protocol offering the services to find and locate information fast and accurate for ad-hoc networks.

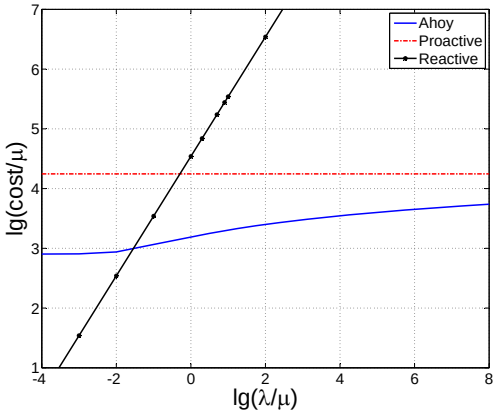
Appendix A

Figures of the Overhead Cost by Ahoy, the Proactive and the Reactive Protocols with Different Paramters

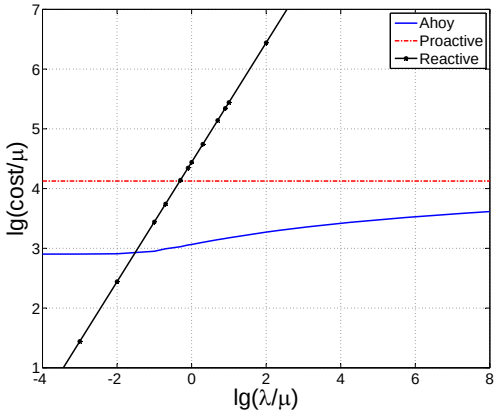


(a) Grid: $d = 3$ (b) Circular: $d = 3$

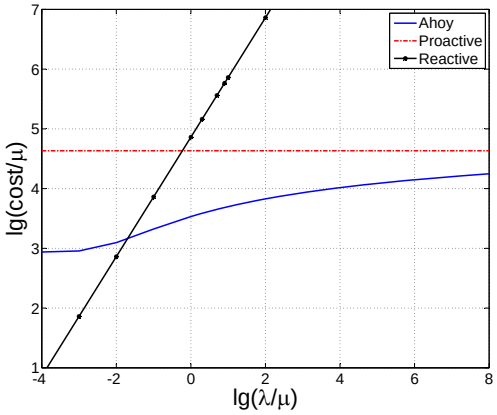
Figure A.1: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 1$



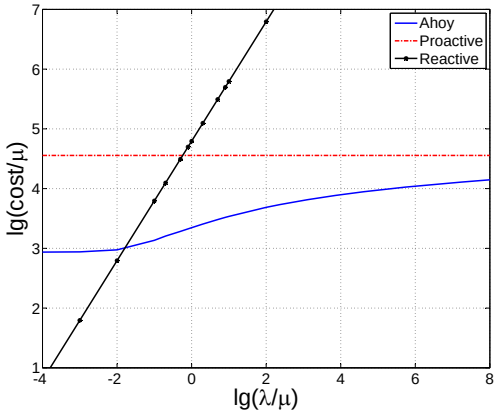
(c) Grid: $d = 4$



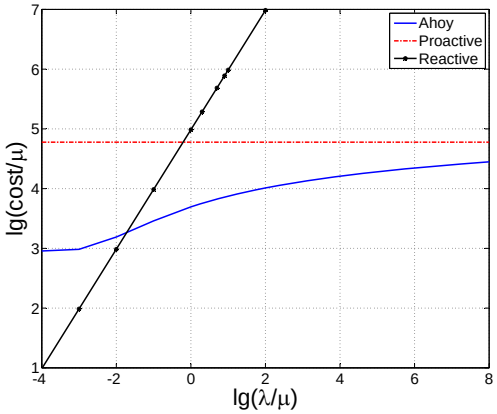
(d) Circular: $d = 4$



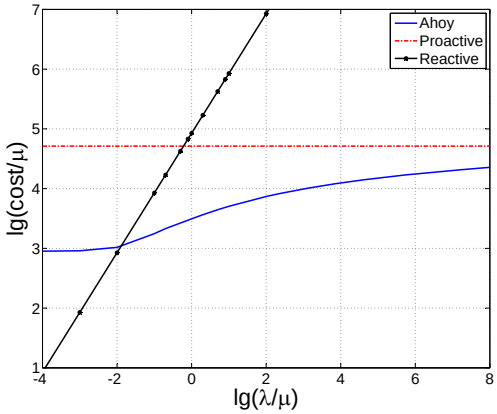
(e) Grid: $d = 6$



(f) Circular: $d = 6$

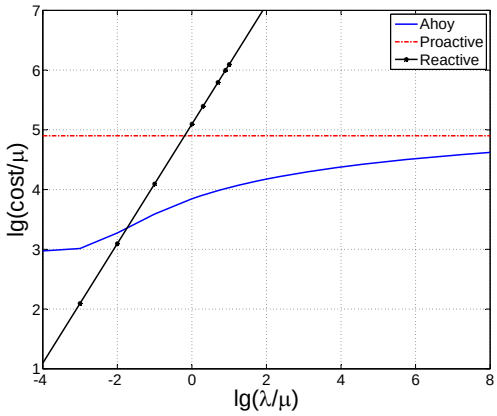


(g) Grid: $d = 7$

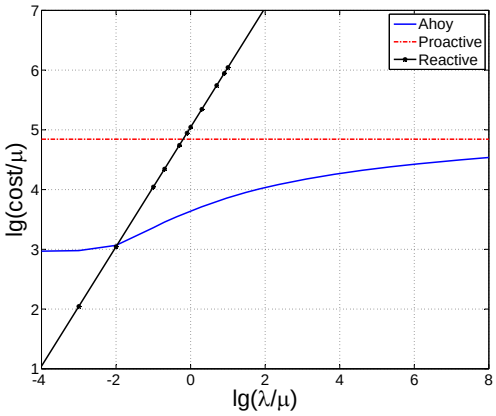


(h) Circular: $d = 7$

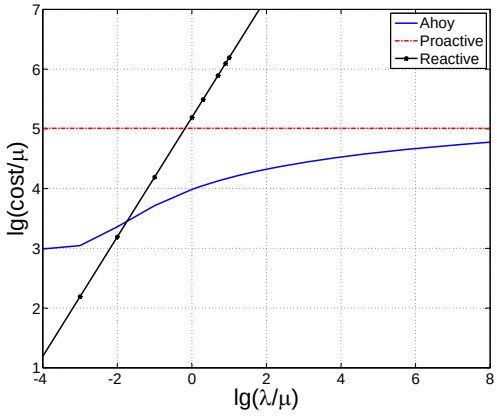
Figure A.1: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 1$



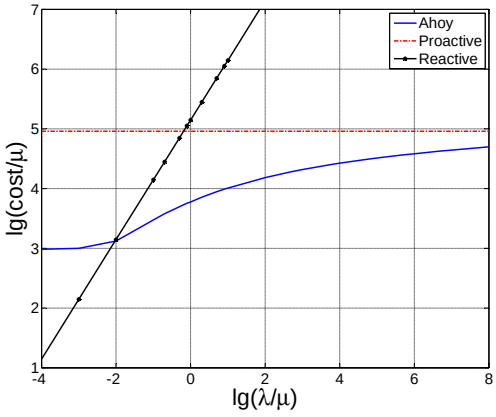
(i) Grid: $d = 8$



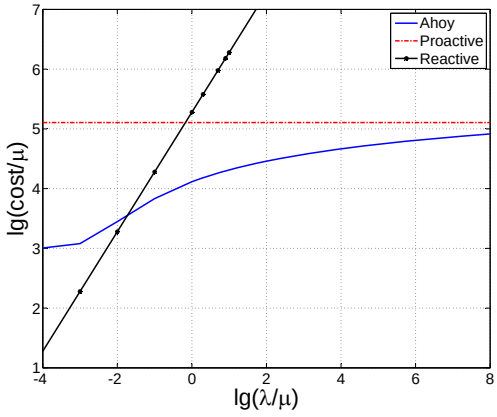
(j) Circular: $d = 8$



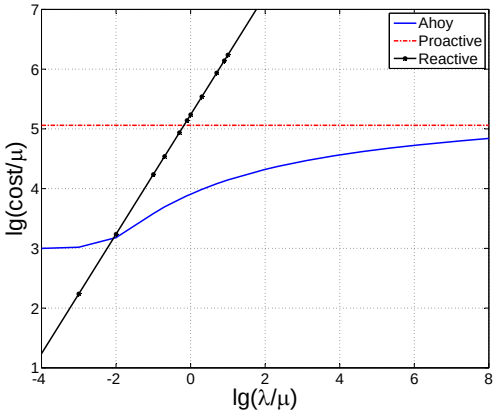
(k) Grid: $d = 9$



(l) Circular: $d = 9$

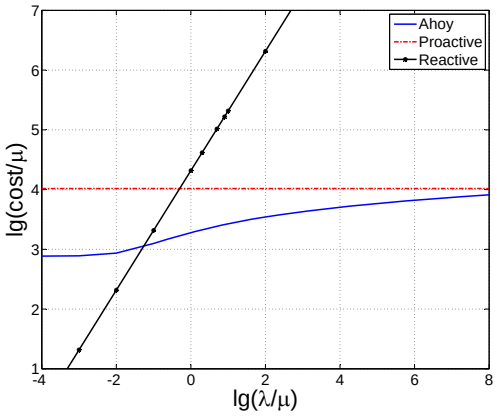


(m) Grid: $d = 10$

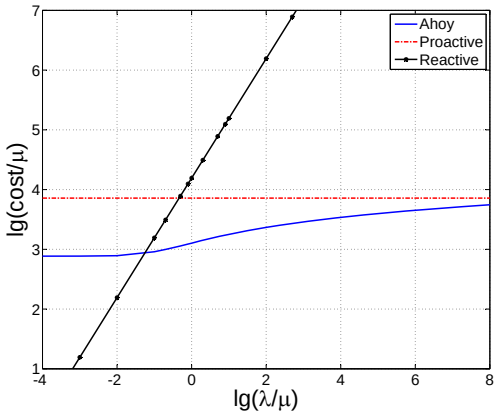


(n) Circular: $d = 10$

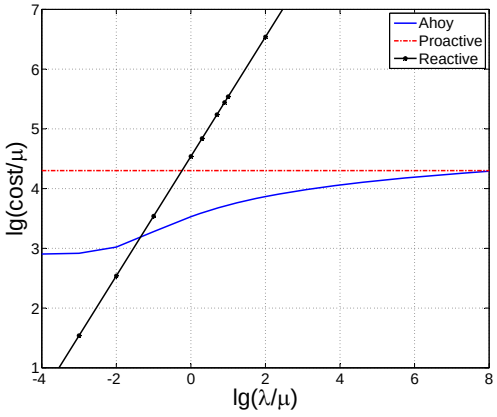
Figure A.1: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 1$



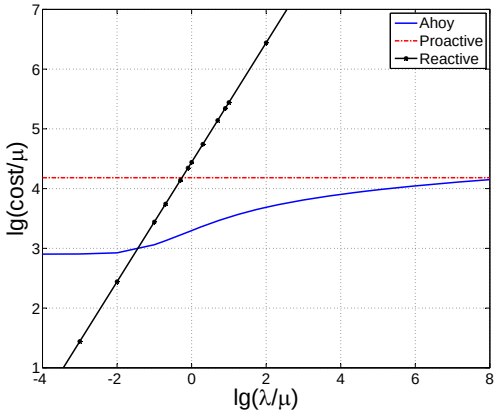
(a) Grid: $d = 3$



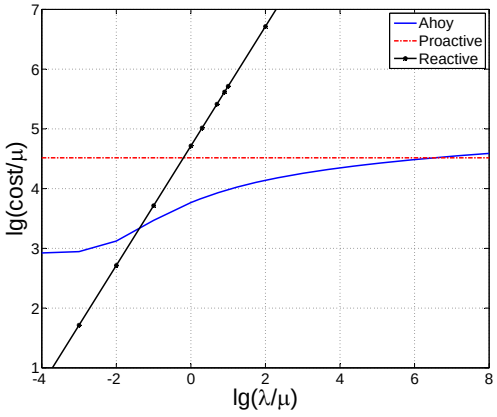
(b) Circular: $d = 3$



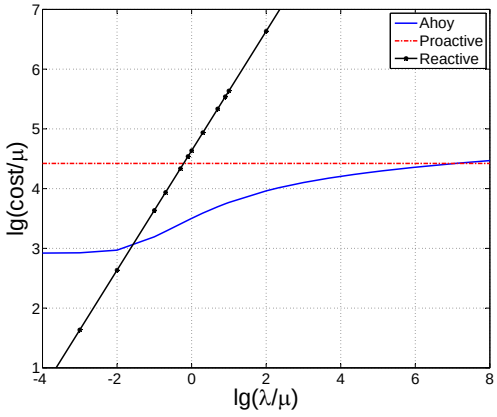
(c) Grid: $d = 4$



(d) Circular: $d = 4$

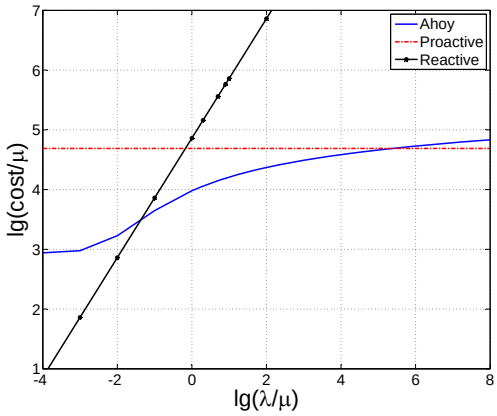


(e) Grid: $d = 5$

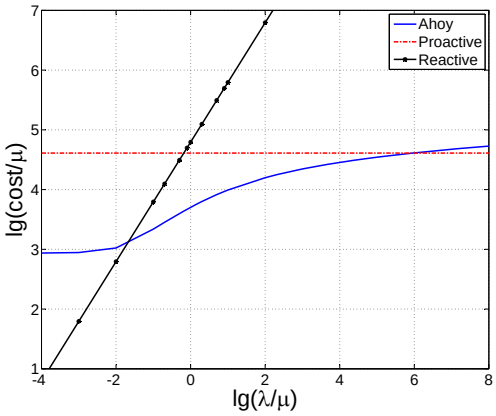


(f) Circular: $d = 5$

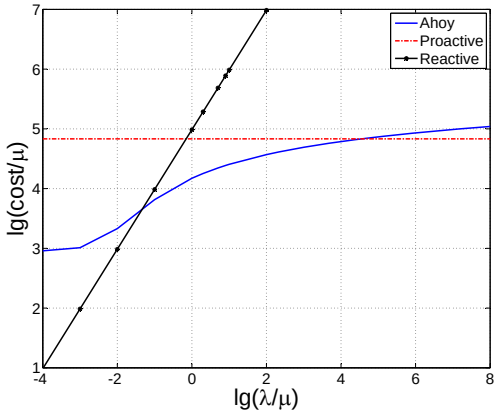
Figure A.2: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 4$



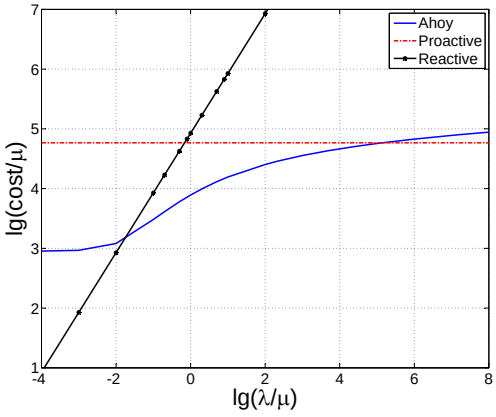
(g) Grid: $d = 6$



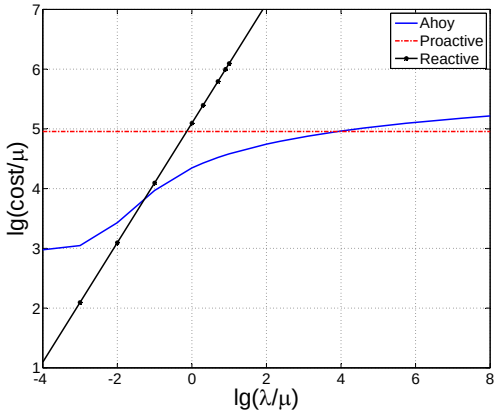
(h) Circular: $d = 6$



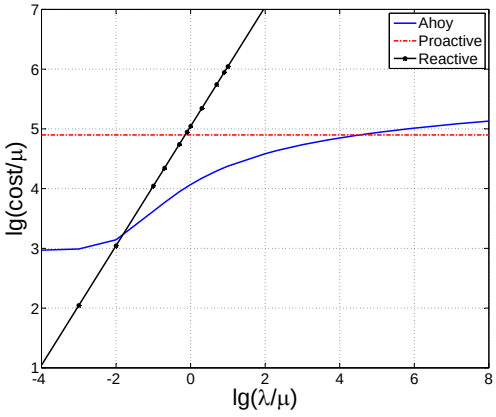
(i) Grid: $d = 7$



(j) Circular: $d = 7$

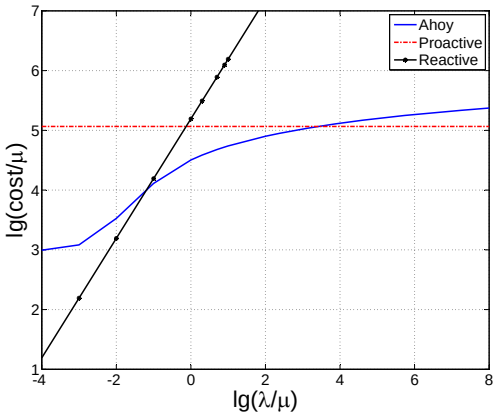


(k) Grid: $d = 8$

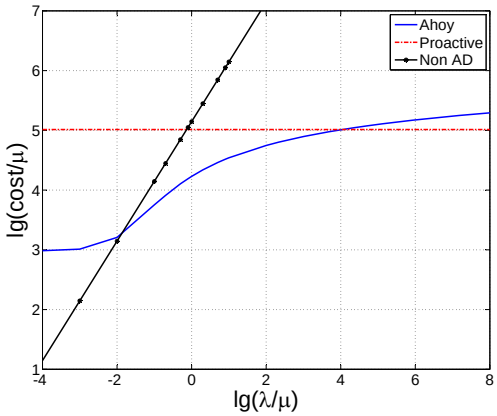


(l) Circular: $d = 8$

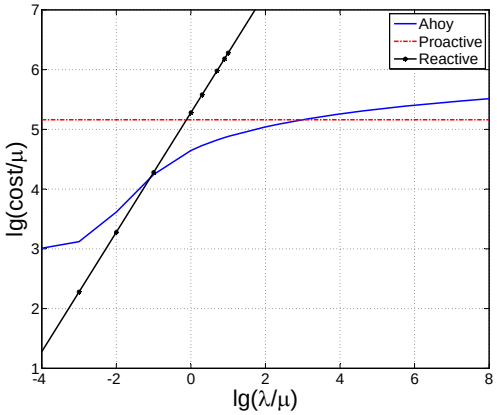
Figure A.2: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 4$



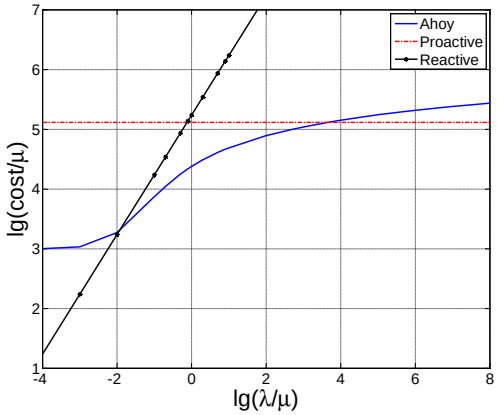
(m) Grid: $d = 9$



(n) Circular: $d = 9$

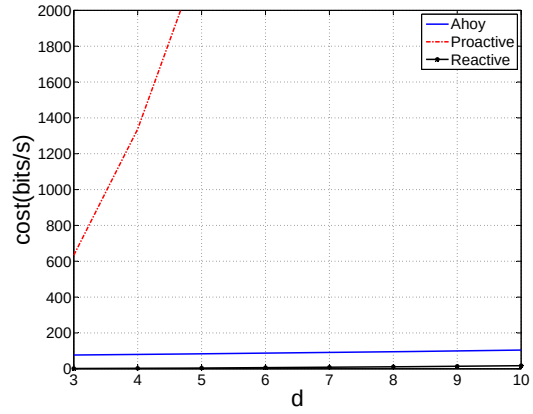


(o) Grid: $d = 10$

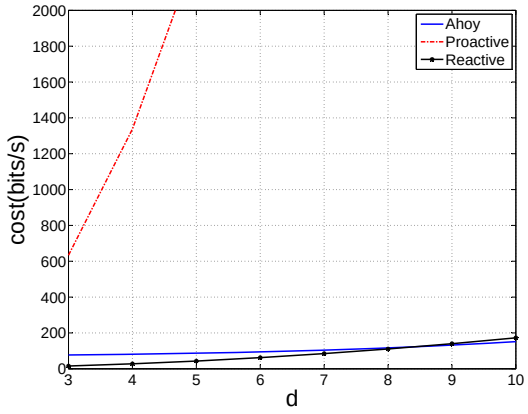


(p) Circular: $d = 10$

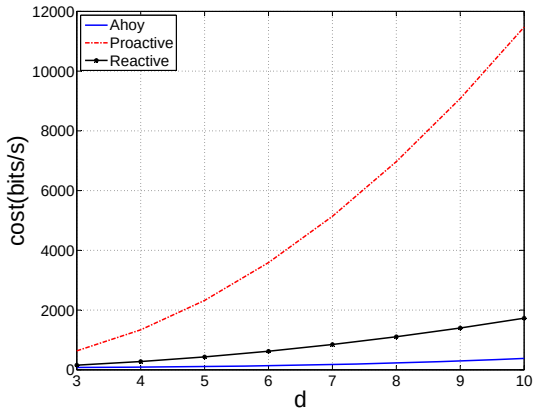
Figure A.2: Overhead costs generated by Ahoy, the proactive and the reactive protocols while varying λ/μ while d varies from 3 to 10 and $s = 4$



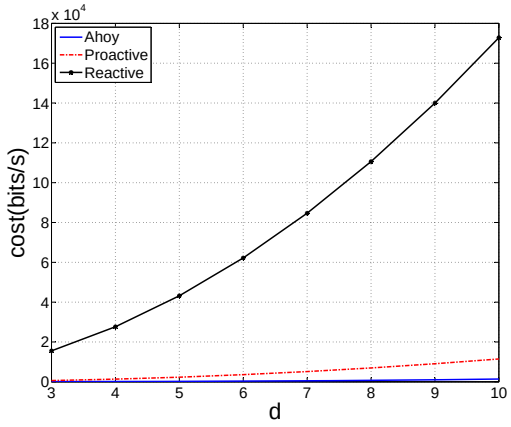
(a) $s = 1$, $\lambda = 0.0001$



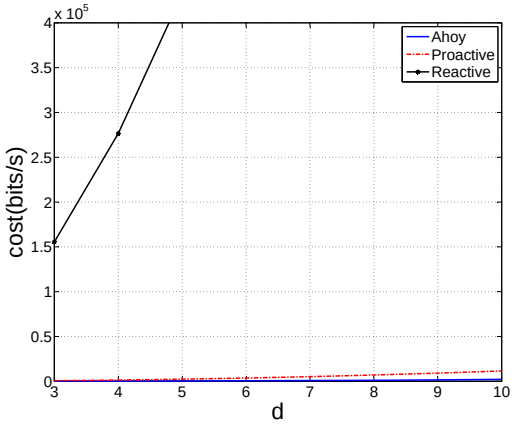
(b) $s = 1$, $\lambda = 0.001$



(c) $s = 1$, $\lambda = 0.01$

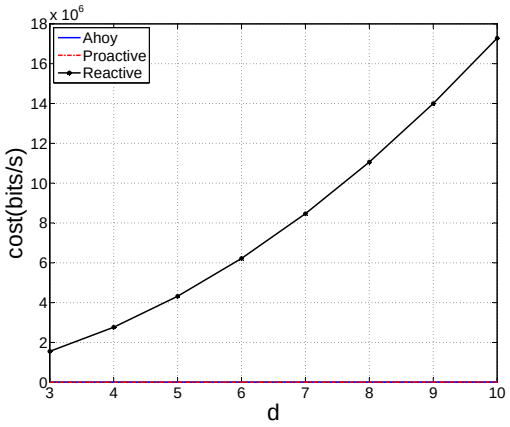


(d) $s = 1$, $\lambda = 1$

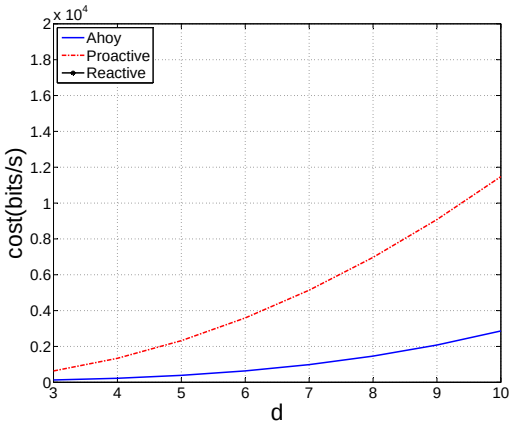


(e) $s = 1$, $\lambda = 10$

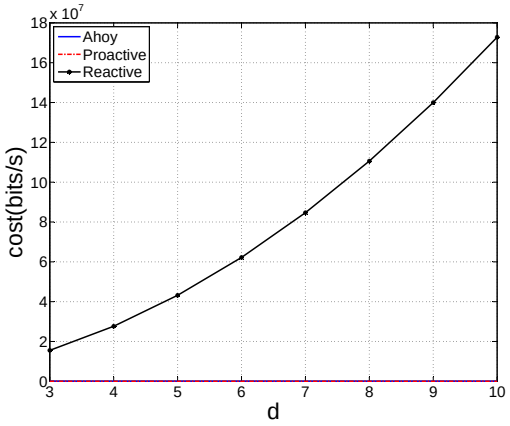
Figure A.3: Overhead cost of Ahoy, the proactive and reactive protocols while varying the ABF depth d, and setting $s = 1$ and $\mu = 0.1$



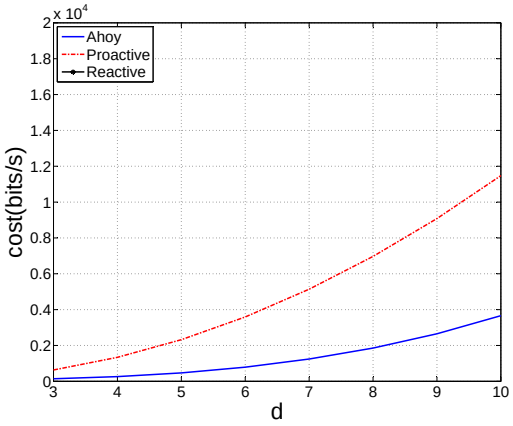
(f) $s = 1$, $\lambda = 100$



(g) $s = 1$, $\lambda = 100$, (focused)

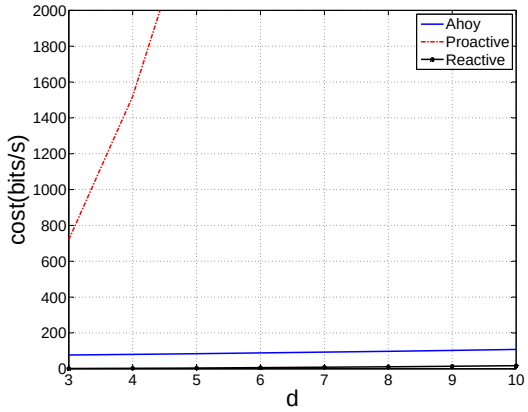


(h) $s = 1$, $\lambda = 1000$

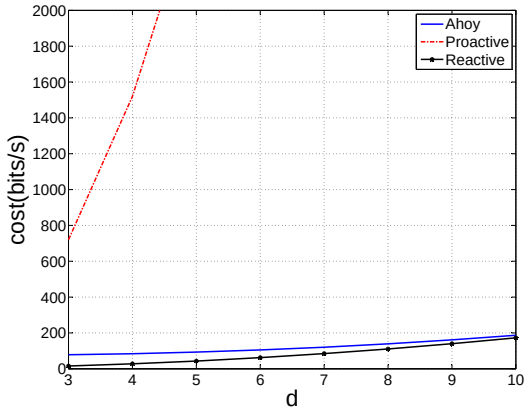


(i) $s = 1$, $\lambda = 1000$, (focused)

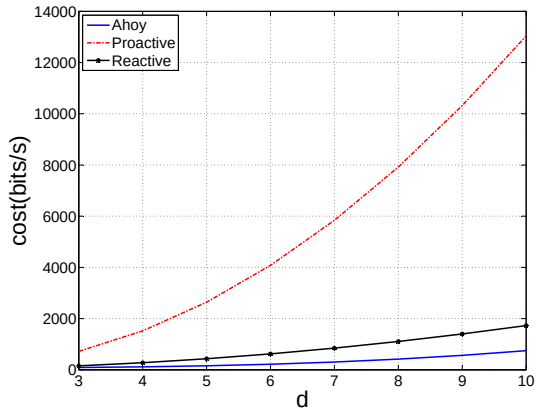
Figure A.3: Overhead cost of Ahoy, the proactive and reactive protocols while varying the ABF depth d, and setting $s = 1$ and $\mu = 0.1$



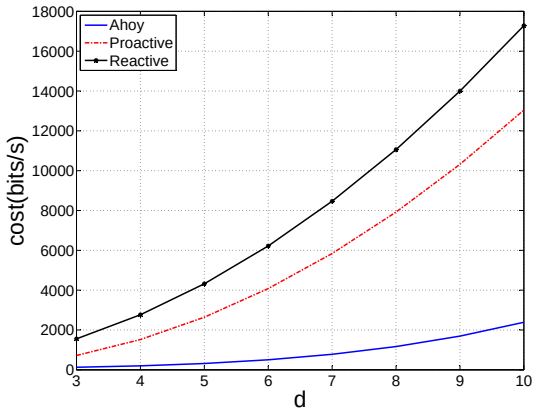
(a) $s = 4$, $\lambda = 0.0001$



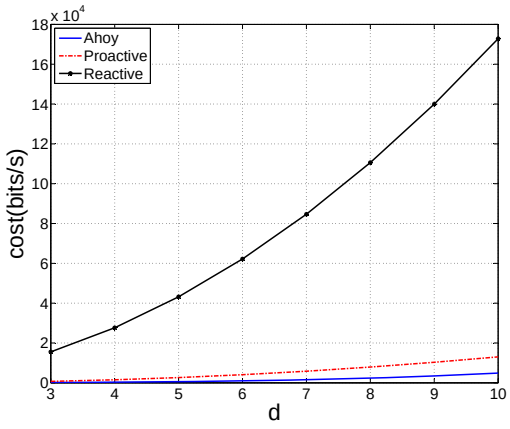
(b) $s = 4$, $\lambda = 0.001$



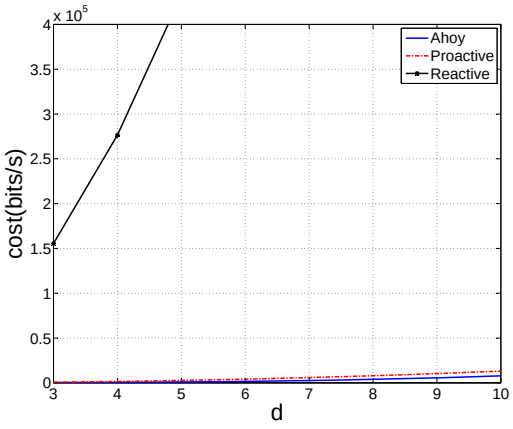
(c) $s = 4$, $\lambda = 0.01$



(d) $s = 4$, $\lambda = 0.1$

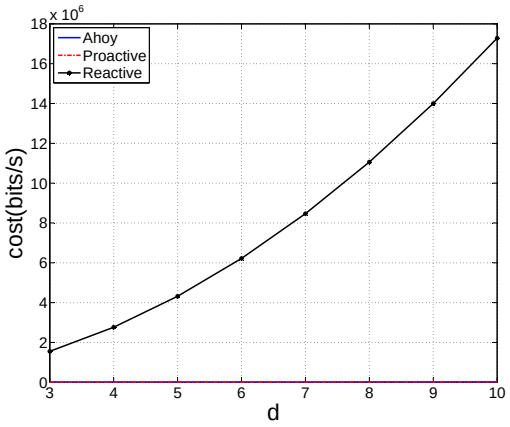


(e) $s = 4$, $\lambda = 1$

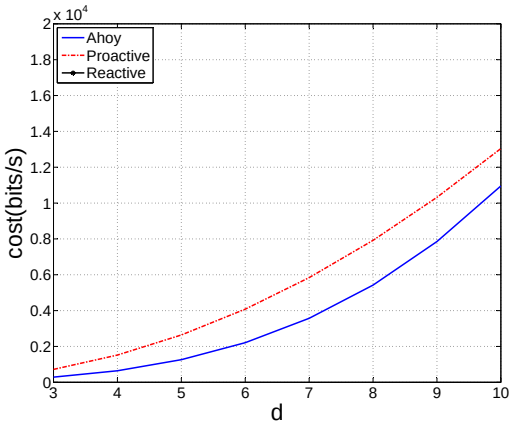


(f) $s = 4$, $\lambda = 10$

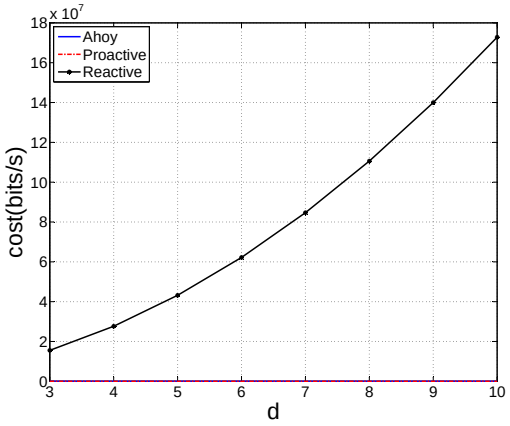
Figure A.4: Overhead cost of Ahoy, the proactive and reactive protocols while varying the ABF depth d, and setting $s = 4$ and $\mu = 0.1$



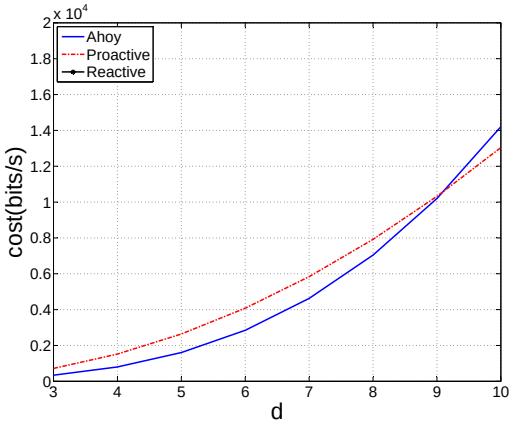
(g) $s = 4$, $\lambda = 100$



(h) $s = 4$, $\lambda = 100$, (focused)

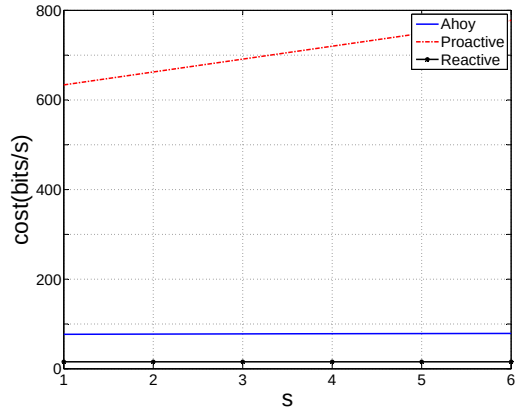


(i) $s = 4$, $\lambda = 1000$

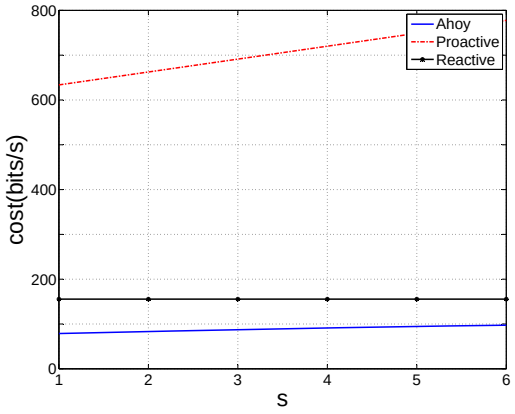


(j) $s = 4$, $\lambda = 1000$, (focused)

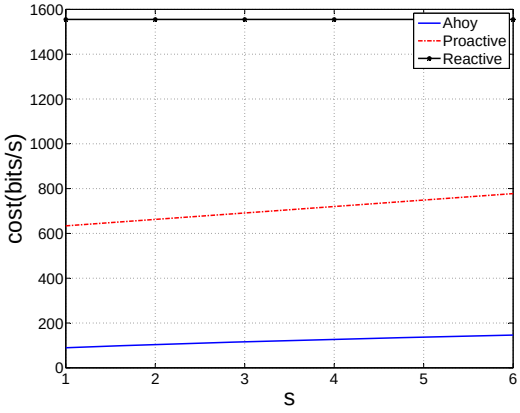
Figure A.4: Overhead cost of Ahoy, the proactive and reactive protocols while varying the ABF depth d, and setting $s = 4$ and $\mu = 0.1$



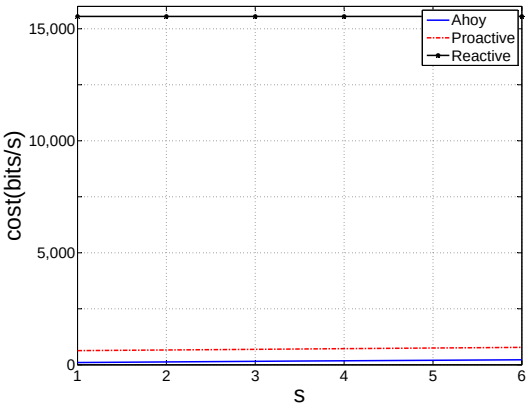
(a) $d = 3$, $\lambda = 0.001$



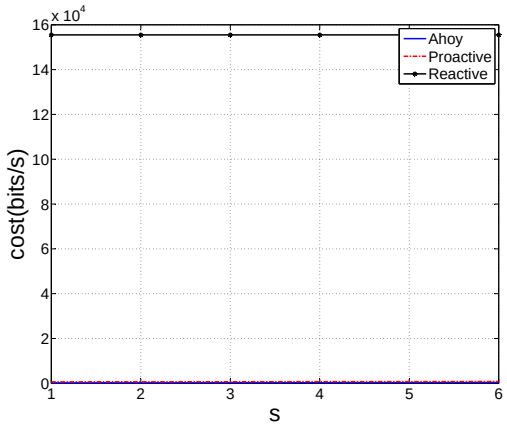
(b) $d = 3$, $\lambda = 0.01$



(c) $d = 3$, $\lambda = 0.1$

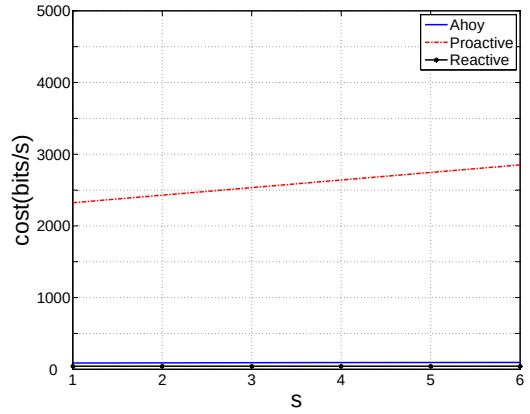


(d) $d = 3$, $\lambda = 1$

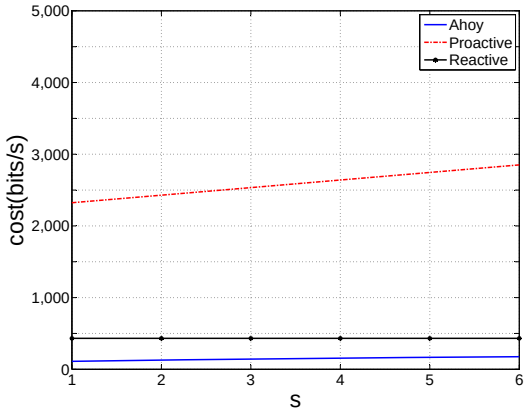


(e) $d = 3$, $\lambda = 10$

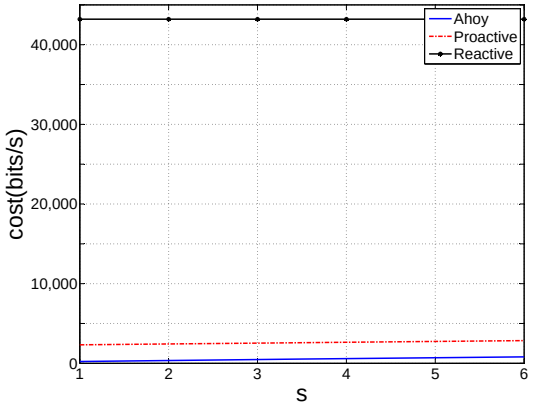
Figure A.5: Impact of the context type density, s, on the overhead cost of Ahoy, the proactive and the reactive protocols, while $d = 3$ and $\mu = 0.1$



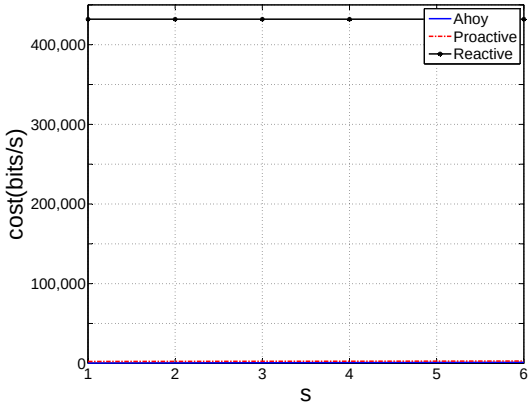
(a) $d = 5$, $\lambda = 0.001$



(b) $d = 5$, $\lambda = 0.01$

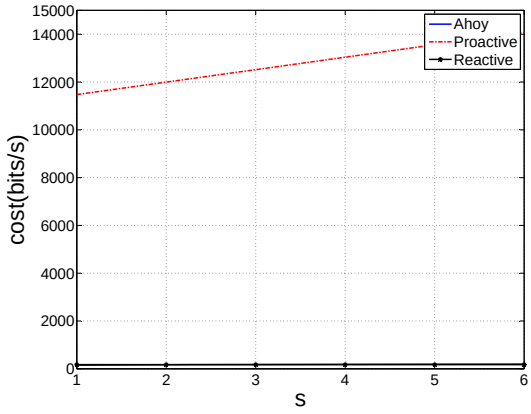


(c) $d = 5$, $\lambda = 1$

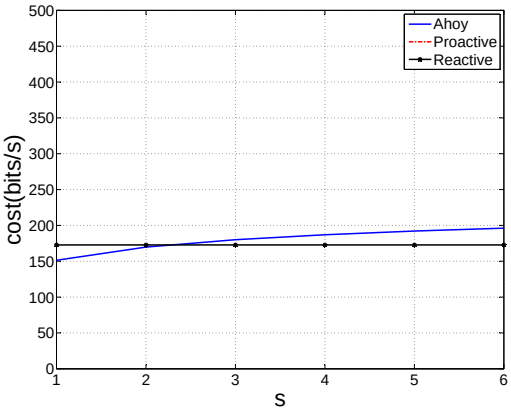


(d) $d = 5$, $\lambda = 10$

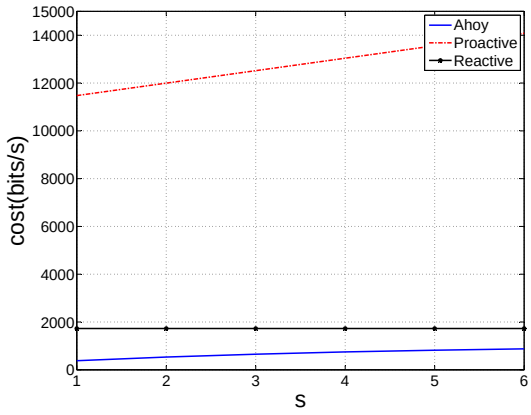
Figure A.6: Impact of the context type density, s, on the overhead cost of Ahoy, the proactive and the reactive protocols, while $d = 5$ and $\mu = 0.1$



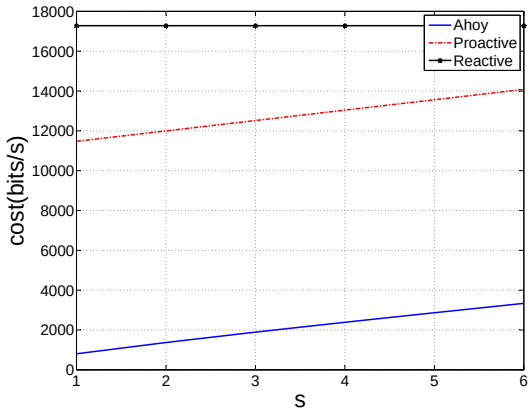
(a) $d = 10$, $\lambda = 0.001$



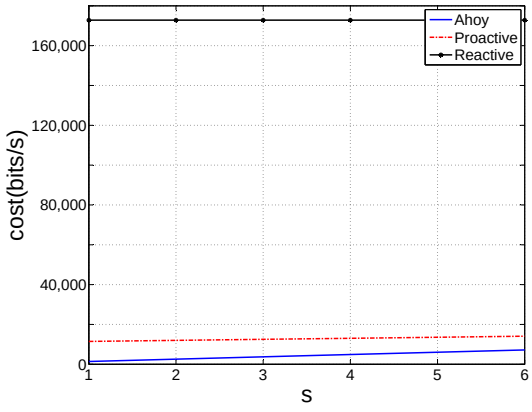
(b) $d = 10$, $\lambda = 0.001$, (focused)



(c) $d = 10$, $\lambda = 0.01$

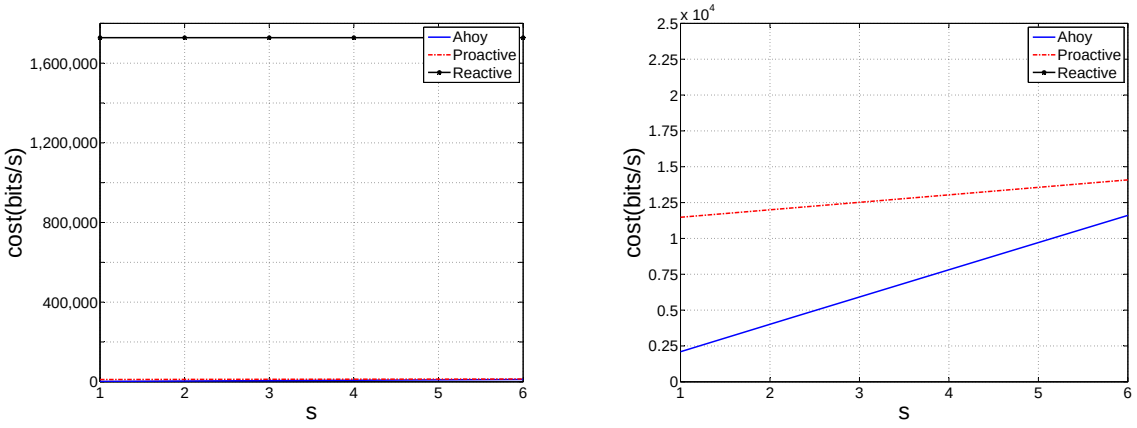


(d) $d = 10$, $\lambda = 0.1$



(e) $d = 10$, $\lambda = 1$

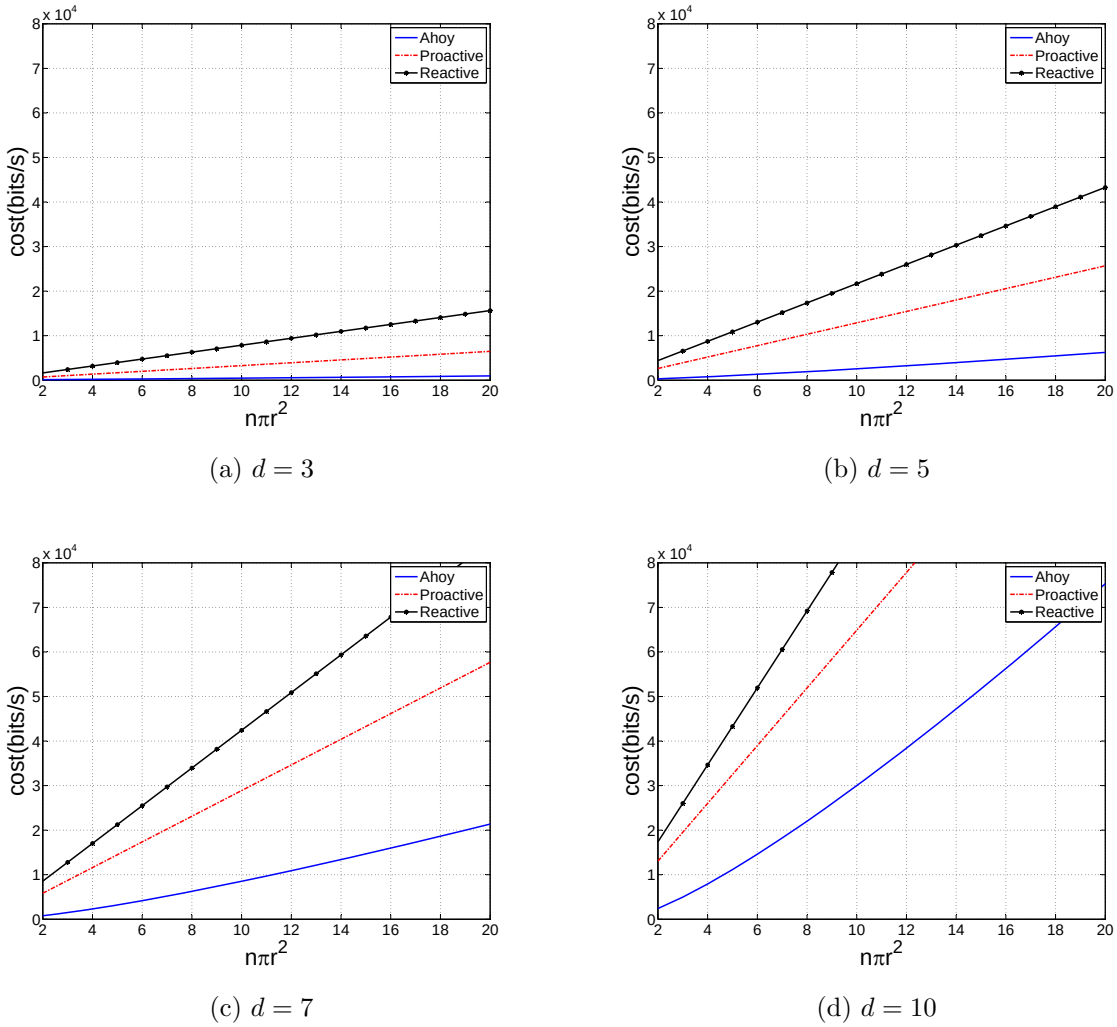
Figure A.7: Impact of the context type density, s, on the overhead cost of Ahoy, the proactive and the reactive protocols, while $d = 10$ and $\mu = 0.1$



(f) $d = 10$, $\lambda = 10$

(g) $d = 10$, $\lambda = 10$, (focused)

Figure A.7: Impact of the context type density, s, on the overhead cost of Ahoy, the proactive and the reactive protocols, while $d = 10$ and $\mu = 0.1$



(a) $d = 3$

(b) $d = 5$

(c) $d = 7$

(d) $d = 10$

Figure A.8: Impact of network density, n, on the overhead cost of Ahoy, the proactive and the reactive protocols, while $s = 4$, $\lambda = 0.1$, $\mu = 0.1$

Appendix B

The Probability Distribution of the Number of Bits Set

In Ahoy, we assume that the total number of context information in range x can be estimated by estimation techniques, such as using historical data, etc. We can obtain the width of filter w and the number of hash functions in use (b) to achieve the minimum overhead cost based on the maximum number of discovery hops d and the number of context information x in range, (see Chapter 4). Hash functions are independent of each other. Each context information is hashed b times by different hash functions. Each time, one bit position is chosen to set to 1. A bit can be chosen multiple times. If the chosen bit has already been set to 1, it stays the same. In total, the bits positions of the filter are chosen $b \times x$ times over the range $[1..w]$. The distribution of the number of 1s set in the filter is the research question here.

We use a discrete time Markov chain (DTMC) to solve the problem. The state space $S = \{0, 1, \cdots, w\}$ represents the total number of 1s set in the filter. The transition between states is taken place whenever one piece of context information is hashed by a hash function. If there are already i bits positions set to 1 in ABFs, the probability for the next bit set in the previous i positions is i/w. The probability for the next bit set into the remaining $(w - i)$ positions is $(w - i)/w$. Therefore, for state $i(0 \leq i \leq w)$, with probability i/w it still stays in the current state i; with probability $(w - i)/w$ it goes to the next state $(i + 1)$. This is illustrated in Figure B.1.

We can translate the Markov chain into a $(w+1) \times (w+1)$ transition probability matrix as follows:

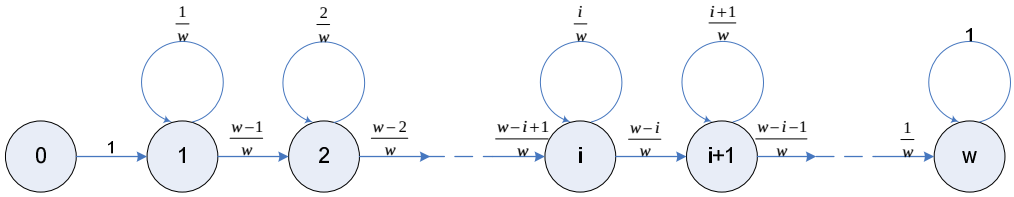


Figure B.1: The states diagram of the total number of 1s set in one filter

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{w} & \frac{w-1}{w} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \frac{2}{w} & \frac{w-2}{w} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & \frac{i}{w} & \frac{w-i}{w} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{B.1}$$

The initial state of the DTMC is given by the vector,

$$p(0) = (1, 0, 0, \cdots, 0) \tag{B.2}$$

For x number of context information, bit positions are set $b \cdot x$ times in total. This corresponds to undertaking $b \cdot x$ steps in the DTMC. The transient state probability distribution at time $b \cdot x$ equals

$$p(bx) = p(0) \cdot \mathbf{P}^{bx}.$$

The ith entry in this vector is denoted by $p_i(bx)$. The discrete random variable Y represents the number of bits that is set in the filter. The probability density function can be given by:

$$f_Y(y) = Pr\{Y = y\} = p_y(bx). \tag{B.3}$$

The cumulative distribution function can be obtained as follows:

$$\begin{aligned} F_Y(y) = Pr\{Y = y\} &= \sum_{i=0}^{y} f_Y(i) = \sum_{i=0}^{y} Pr\{Y = i\} \\ &= \sum_{i=0}^{y} p_i(bx), \end{aligned} \tag{B.4}$$

and equals the sum of the first y entries of the state probability distribution at time bx.

Bibliography

[1] W. Adjie-Winoto, E. Schwarts, H. Balakrishnan, and J. Lilley. The design and implementation of an international naming system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles, SOSP'99*, Charleston, United States, December 1999.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks Elsevier Journal*, 38(4):393–422, March 2002.

[3] Y. an Huang and W. Lee. Hotspot-based traceback for mobile ad-hoc networks. In *Proc. of the 4th ACM Workshop on Wireless Security*, Cologne, Germany, September 2005.

[4] F. Anjum and P. Mouchtaris. *Security for Wireless Ad Hoc Networks*. John Wiley and Sons Ltd, 2007.

[5] C. Bettstetter. On the minimum node degree and connectivity of a wireless multihop networks. In *Proc. of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing, MOBIHOC'02*, EPF Lausanne, Switzerland, June 2002.

[6] B. H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Bluetooth SIG. Bluetooth core specification 3.0 + HS. http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/, April 2009.

[8] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: a survey. *Internet Mathematics*, 1(4):485–509, 2003.

[9] G. Brose, A. Vogel, and K. Duddy. *Java Programming with CORBA*. John Wiley and Sons, Inc, 2001.

[10] S. A. Camtepe and B. Yener. Key distribution mechanisms for wireless sensor networks: a survey. Technical report, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, US, 2005.

[11] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Computer and System Sciences*, 18(2):143–154, 1979.

[12] D. Chakraborty, A. Joshi, Y. Yesha, and T. Finin. Gsd: a novel group-based service discovery protocol for MANETs. In *Proc. of the 4th IEEE International Conference on Mobile and Wireless Communications Networks, MWCN 2002*, Stockholm, Sweden, September 2002.

[13] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, and A. Fikes. Bigtable: A distributed storage system for structured data. In *Proc. of the 7th Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, United States, November 2006.

[14] I. Chlamtac, M. Conti, and J. J. Liu. Mobile ad hoc networking: Imperatives and challenges. *Ad Hoc Networks*, 1(1):13–64, July 2003.

[15] T. Clausen and P. Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, Project Hipercom, INRIA, October 2003.

[16] A. Conta and S. Deering. Internet control message protocol (ICMPv6) for the internet protocol version 6 (IPv6) specification. RFC 2463, Lucent and Cisco Systems, December 1998.

[17] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, August 2003.

[18] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proc. of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking, MobiCom 99*, Seattle, United States, August 1999.

[19] N. Daswani, C. Kern, and A. Kesavan. *Foundations of Security: What Every Programmer Needs To Know*. Apress, 2007.

[20] Debian webpage. http://www.debian.org/, March 2011.

[21] M. Degermark, B. Nordgren, and S. Pink. IP header compression. RFC 2507, Lulea University of Technology, February 1999.

[22] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, December 2001.

[23] N. Ferugson. Michael: an improved MIC for 802.11 WEP. IEEE 802.11 doc 02-020r0, http://grouper.ieee.org/groups/802/11/Documents/DocumentHolder/2-020.zip, January 2002.

[24] C. Frank and H. Karl. Consistency challenges of service discovery in mobile ad hoc networks. In *Proc. of the 7th International Symposium on Modeling, Analysis and Simulation of Wireless and MObile Systems, MSWiM 2004*, Venice, Italy, November 2004.

[25] Freeband AWARENESS webpage. http://www.freeband.nl/project.cfm?id=494, March 2011.

[26] J. Gao and P. Steenkiste. Rendezvous points-based scalable content discovery with load balancing. In *Proc. of Fouth International Workshop on Network Group Communication*, Boston, USA, October 2002.

[27] P. T. Goering and G. J. Heijenk. Service discovery using Bloom filters. In *Proc. of 12th Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 2006.

[28] P. T. Goering, G. J. Heijenk, B. Haverkort, and R. Haarman. Effect of mobility on local service discovery in ad-hoc. In *Proc. of Performance Engineering Workshop 2007*, Berlin, Germany, September 2007.

[29] L. L. Gremillion. Designing a Bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, 1982.

[30] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service discovery protocol version 2. RFC 2608, Sun Microsystems and @Home Network and Vinca Corporation, June 1999.

[31] R. Haarman. Ahoy: A proximity-based discovery protocol. Master's thesis, University of Twente, the Netherlands, January 2007.

[32] R. Haarman, F. Liu, P. T. Goering, and G. J. Heijenk. Proof-of-Concept implementation of the ahoy discovery protocol in ad-hoc networks. CTIT technical report, Universiy of Twente, Centre for Telematics and Information Technology, the Netherlands, May 2011.

[33] B. R. Haverkort. *Performance of Computer Communciation Systems: a Model-based Approach*. John Wiley and Sons Ltd, 1998.

[34] R. Hekmat and P. V. Mieghem. Degree distribution and hopcount in wireless ad-hoc networks. In *Proc. of the 11th IEEE International Conference on Networks, ICON2003*, Sydney, Austrilia, September 2003.

[35] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - a service discovery protocol for ad-hoc networks. In *Proc. of the 3rd IEEE Conference on Wireless Communication Networks, WCNC 2003*, New Orleans, USA, March 2003.

[36] A. Helmy. *Resource Management in Wireless Networking*, volume 16, chapter Efficient Resource Discovery in Wireless Ad-hoc Networks: Contents Do Help, pages 419–471. Springer US, 2005.

[37] J. Hoebeke, I. Moerman, B. Dhoedt, and P. Demeester. Analysis of decentralized resource and service discovery mechanisms in wireless multi-hop networks. In *Proc. of the 3rd international Conference on Wired/Wireless Internet Communications, WWIC2005*, Xanthi, Greece, May 2005.

[38] IEEE Computer Society LAN/MAN Standards Committee. IEEE standard for information technologytelecommunications and information exchange between systemslocal and metropolitan area networksspecific requirements, part 11: Wireless lan medium access control (MAC) and physical layer (PHY) specifications. Technical report.

[39] IEEE SA. 802.11n standard. http://standards.ieee.org/getieee802/download/802.11n-2009.pdf, 2009.

[40] IEEE standards association. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. http://standards.ieee.org/getieee802/802.11.html, 1999.

[41] IEEE standards association. Draft amendment to standard for telecommunications and information exchange between systems - LAN/MAN specification requirements - part 11: Wireless medium access control (MAC) and physical layer (PHY) specifications: Medium access control (MAC) security enhancements. IEEE Std 80211i/D7.0, October 2003.

[42] IEEE standards association. 802.11n-2009 IEEE standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (MAC) and physical layer

(PHY) specifications amendment 5: Enhancements for higher throughput. http://ieeexplore.ieee.org/servlet/opac?punumber=5307291, 2009.

[43] D. Johnson, C. Perkins, and J. Arkko. Mobility support in IPv6. RFC 3775, Rice Univeristy, Nokia Research Center and Ericsson, June 2004.

[44] M. R. Jongerden. *Model-based energy analysis of battery powered systems*. PhD thesis, University of Twente, the Netherlands, December 2010.

[45] M. Klein, B. Konig-Ries, and P. Obreiter. Service rings - a semantic overlay for service discovery in ad hoc networks. In *Proc. of the 14th International Workshop on Database and Expert Systems Applications, DEXA 2003*, Prague, Czech Republic, September 2003.

[46] T. Kosch, C. Adler, S. Eichler, C. Schroth, and M. Strassberger. The scalability problem of vehicular ad hoc networks and how to solve it. *IEEE Wireless Communications Magazine*, 13(5):22–28, October 2006.

[47] U. C. Kozat and L. Tassiulas. Service discovery in mobile ad hoc networks: an overall perspective on architectural choices and network layer support issues. *Ad Hoc Networks*, 2(1):23–44, June 2003.

[48] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[49] V. Lenders, M. May, and B. Plattner. Service discovery in mobile ad hoc networks: A field theoretic approach. *Pervasive and Mobile Computing*, 1(1):343–370, March 2005.

[50] F. Liu, P. T. Goering, and G. J. Heijenk. Modeling service discovery in ad-hoc networks. In *Proc. of the 4th ACM International Workshop on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks, PE-WASUN 2007*, Chania, Greece, October 2007.

[51] F. Liu and G. J. Heijenk. Context discovery using attenuated Bloom filters in ad-hoc networks. In *Proc. of the 4th Wired/Wireless Internet Communications, WWIC06*, Bern, Switzerland, May 2006.

[52] F. Liu and G. J. Heijenk. Context discovery using attenuated Bloom filters in ad-hoc networks. *Journal of Internet Engineering*, 1(1):49–58, 2007.

[53] F. Liu and G. J. Heijenk. Dynamic connectivity analysis of abf-based ad-hoc networks. In *Proc. of the 1st IFIP Wireless and Mobile Networking Conference*, Toulouse, France, September 2008.

[54] F. Liu and G. J. Heijenk. On the impact of network dynamics on a discovery protocol for ad-hoc networks. *International Journal of Business Data Communications and Networking*, 5(3):16–34, 2009.

[55] J. Liu, D. Sacchetti, F. Sailhan, and V. Issarny. Group management for mobile ad-hoc networks: Design, implementation and experiment. In *Proc. of the 6th IEEE International Conference on Mobile Data Management, MDM '05*, Ayia Napa, Cyprus, May 2005.

[56] R. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. PhD thesis, University of Twente, the Netherlands, November 2008.

[57] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[58] B. T. Mohamed, M. Abdelaziz, and E. Pouyoul. Project JXTA: A loosely-consistent DHT rendezvous walker, May 2003.

[59] U. Mohan, K. C. Almeroth, and E. M. Belding-Royer. Scalable service discovery in mobile ad hoc networks. *Lecture Notes in Computer Science*, 3042/2004:137–149, 2004.

[60] J. K. Mullin. A second look at Bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.

[61] P. Mutaf and C. Castelluccia. Compact neighbor discovery. In *Proc. IEEE INFOCOM 2005*, Miami, March 2005.

[62] M. Nidd. Service discovery in deapspace. *IEEE PCM*, 8(4):39–45, August 2001.

[63] OPNET modeler software. http://www.opnet.com/products/modeler, March 2011.

[64] E. Papapetrou, E. Pitoura, and K. Lillis. Speeding-up cache lookups in wireless ad-hoc routing using Bloom filters. In *Proc. of the 16th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2005*, Berlin, Germany, September 2005.

[65] M. Penrose. *Random Geometric Graphs*. Oxford University Press Inc, 2003.

[66] C. Perkins. IP mobility support for IPv4. RFC 3344, Nokia Research Center, August 2002.

[67] R. Ramanathan and J. Redi. A brief overview of ad hoc networks: challenges and directions. *Communications Magazine, IEEE*, 40(5):20–22, May 2002.

[68] O. Ratsimor, D. Chakraborty, A. Joshi, and T. Finin. Allia: Alliance-based service discovery for ad-hoc environments. In *Proc. of 2nd ACM Mobile Commerce Workshop, MC 2002*, Atlanta, USA, September 2002.

[69] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, New York, United States, June 2005.

[70] Ruby online home. http://www.ruby-lang.org/, March 2011.

[71] J. Schiller. *Mobile Communications*. Addision Wesley, 2000.

[72] M. Sipser. *Introduction to the Theory of computation, Second Edition*. Thomson Course Technology, 2006.

[73] D. H. Steiberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, 2005.

[74] Sun Microsystems. DJ - Jini disocvery and join specification v3.0. http://java.sun.com/products/jini/2.1/doc/specs/html/discovery-spec.html, September 2006.

[75] V. Sundramoorthy, H. Scholten, P. Jansen, and P. Hatel. Service discovery at home. In *Proc. of the 4th International Conference on Information, Communications and Signal Processing and 4th IEEE Pacifi-Rim conference*, Singapore, December 2003.

[76] Tcpdump public repository. http://www.tcpdump.org/, March 2011.

[77] The Salutation Consortium. Salutation architecture specification version 2.0c. http://www.salutation.org, June 1999.

[78] UPnP Forum. UPnP device architecture version 1.0, June 1999.

[79] User Mode Linux. http://www.usermodelinux.org/, March 2011.

[80] A. Varshavsky, B. Reid, and E. de Lara. A cross-layer approach to service discovery and selection in MANETs. In *Proc. of Mobile Ad-hoc and Sensor System Conference, MASS 2005*, Washington, USA, November 2005.

[81] B. Wu, J. Chen, J. Wu, and M. Cardei. *Wireless Network Security*, chapter A Survey of Attacks and Countermeasures in Mobile Ad Hoc Networks, pages 103–135. Springer US, Florida Atlantic University, US, 2007.

[82] XML-RPC webpage. http://www.xmlrpc.com/, March 2011.

[83] Y. Yang, H. Hassanein, and A. Mawji. Efficient service discovery forwireless mobile ad hoc networks. In *Proc. of IEEE International Conference on Computer Systems and Applications, AICCSA 2006*, Dubai, Sharjah, March 2006.

[84] L. Zhou and Z. J. Haas. Securing ad hoc networks. *IEEE Network*, 13(1):24–30.

[85] F. Zhu, M. Mutka, and L. Ni. Splendor: A secure, private, and location-aware service discovery protocol supporting mobile services. In *Proc. of the First IEEE International Conference on Pervasive Computing and Communications, PerCom 2003*, Texas, USA, March 2003.

About the Author

Fei Liu was born in Changzhou, China, on September 17, 1980. She obtained her B.Sc. degree in Computer Science (with honor) from Jiangsu Polytechnic University, China in 2002. She then came to the Netherlands and started her master study in Telematics in the University of Twente. In 2004 she was awarded her Master degree with the master thesis titled Design and Develop an Interference-based Routing Algorithm for Cellular Networks. Since then she has been conducting her Ph.D. research on Context Discovery in Ad-hoc Networks in the same group, and as part of Freeband AWARENESS project. From 1 June 2009, Fei Liu has joined the Center for Transport Studies (CTS) in the same university. She has been working on the development and simulation of Cooperative Adaptive Cruise Control (CACC) systems, as part of Connect & Drive project. Her main topics of interest are context discovery, mobile and wireless networking, intelligent transportation systems.

Her publications are listed below in chronological order:

- Haarman, R., Liu, F., Goering, P., and Heijenk, G. (2011). *Proof-of-Concept Implementation of the Ahoy Discovery Protocol in Ad-hoc Networks.* Technical report TR-CTIT-11-10, Centre for Telematics and Information Technology Universiy of Twente, Enschede.

- Pueboobpaphan, R., Liu, F. and Arem, B. van (2010). *The Impacts of a Communication based Merging Assistant on Traffic Flows of Manual and Equipped Vehicles at an On-ramp Using Traffic Flow Simulation.* In: 13th International Conference on Intelligent Transportation Systems, September 19-22-2010 in Madeira Island, Portugal. (DVD) (pp. 1468-1473). Funchai: ITS (ISBN 978-142447657-2).

- Liu, F., Pueboobpaphan, R., Van Arem, B. (2009). *Assessment of Traffic Impact on Future Cooperative Driving Systems: Challenges and Considerations.* International Workshop on Communication Technologies for Vehicles, October, Saint-Petersburg, Russia.

- Liu, F. and Heijenk, G.J. (2009) *On the impact of network dynamics on a discovery protocol for ad-hoc networks.* International Journal of Business Data Communications and Networking, 5 (2). pp. 16-34.

- Pawar, P. and Boros, H. and Liu, F. and Heijenk, G.J. and van Beijnum, B.J.F. (2009) *Bridging Context Management Systems in the ad hoc and mobile environments.* In: IEEE Symposium on Computers and Communications, 2009. ISCC 2009., 5-8 July 2009, SOusse, Tunisia. pp. 882-888. IEEE Computer Society Press.

- Hesselman, C.E.W. and Benz, H.P. and Pawar, P. and Liu, F. and Wegdam, M. and Wibbels, M. and Broens, T.H.F. and Brok, J. (2008) *Bridging context management systems for different types of pervasive computing environments.* In: Proceedings of the First International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, 15-17 February 2008, Innsbruck, Austria. pp. 1-8. ACM International Conference Proceedings serie 278. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering.

- Liu, F. and Heijenk, G.J. (2008) *Dynamic connectivity analysis of ABF-based ad-hoc networks.* In: Proceedings of the 1st IFIP Wireless and Mobile Networking Conference, 30 Sep - 02 Oct 2008, Toulouse, France. pp. 407-420. Springer Verlag.

- Liu, F. and Goering, P.T.H. and Heijenk, G.J. (2007) *Modeling service discovery in ad-hoc networks.* In: Proceedings of the Fourth ACM International Workshop on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks, PE-WASUN 2007, 22 - 26 Oct 2007, Chania, Greece. pp. 9-16. Association for Computing Machinery.

- Liu, F. and Heijenk, G.J. (2007) *Context discovery using attenuated Bloom filters in ad-hoc networks.* Journal of Internet Engineering, 1 (1). pp. 49-58.

- Heijenk, G.J. and Liu, F. (2006) *Interference-based routing in multi-hop wireless infrastructures.* Computer Communications, 29 (13-14). pp. 2693-2701. *** ISI Impact 0,933 ***

- Liu, F. and Heijenk, G.J. (2006) *Context discovery using attenuated Bloom codes: model description and validation.* Technical Report TR-CTIT-06-09, Centre for Telematics and Information Technology University of Twente, Enschede.

- Liu, F. and Heijenk, G.J. (2006) *Context discovery using attenuated Bloom filters in ad-hoc networks.* In: Proceedings 4th International Conference on Wired/Wireless Internet Communications, WWIC 2006, May 9-12, 2006, Bern, Switzerland. pp. 13-25. Lecture Notes in Computer Science 3970. Springer-Verlag.

- Heijenk, G.J. and Liu, F. (2005) *Interference-based routing in multi-hop wireless infrastructures.* In: Proceedings Wired/Wireless Internet Communications: Third International Conference, WWIC 2005, 11 - 13 May 2005, Xanthi, Greece. pp. 117-127. Lecture Notes in Computer Science 3510 / 2005.

Acknowledgements

This book does not only record the scientific research I have done, but also the precious experience I had in the past couple of years. I would like to express my gratitude to all those who have helped me during the PhD trajectory.

Dear Geert, I am very lucky to have you as my daily supervisor. From the master's assignment to the PhD research, you have taught me a lot. I learned how to do solid scientific research and how to organize and present my work. I believe throughout the entire PhD training, the most valuable outcome is the proper manner of conducting research, the discipline, and the rigorous attitude towards science. I will benefit from your guidance and influence for my whole life. Geert, you are not only a wonderful supervisor at work, but also a great mentor and friend in life. In the most difficult period of my life, your offered me many generous help and support. Without you, this thesis could not have reached this far.

I would also like to thank Boudewijn Haverkort for his helpful insights and great advice. I have been constantly inspired by every discussion we had.

It is my honor to have Prof. dr. Marilia Curado, Prof. dr. ir. Erik R. Fledderus, Prof. dr. ir. Sonia Heemstra de Groot, Prof. dr. Hans van den Berg, and Prof. dr. ir. Kees C.H. Slump as members of my graduation committee. I thank each of them for accepting the invitation and reviewing my work.

The design and development of the Ahoy discovery protocol is a team work. I have collaborated closely with Patrick Goering and Robbert Haarman during the first couple of years. I am grateful for all fruitful discussions and collaborations. It was a great fun to work with you two! Thank you Robbert for carefully reading the draft version of the thesis.

I would like to thank Marijn Jongerden, Anne Remke, and Pieter-Tjerk de Boer for their help with mathematic proofs.

I would also like to thank all the colleagues from the AWARENESS project.

It was a great experience to work in a Dutch national project and cooperate with colleagues with different expertise.

Special thanks to Tom Thomas who has given me lots of inspiration at the final phase and spent days and nights to improve the language in this thesis. I would also like to thank Wouter Hermelink for the cover design of this book.

Friendship is especially precious when you are in trouble. My dear friends, thank you all very much for your selfless help and support. Without your warm hugs and encouragement, it would have been very difficult to complete this work and continue my life in Holland. Thank you very much Yimeng, Yuanliang, Silvia, Geert, Didem, Semih, Albert, Yuanqing, Lei, Shi, Anna, Hailiang, Pravin, Desi, Marijn, Anne, Georgios, Aiko, Tiago, Ramin, Tatiana, Lucia, Boudewijn, Wouter, Idilio, Martijn. I am thankful to all the people who helped me to overcome those difficulties and make me grow up.

I would also like to thank all my colleagues in the Center of Transportation Study in University of Twente for making a very "gezellig" environment and make me feel part of the team. I especially wish to thank Eric van Berkum, Bart van Arem, and Marieke Martens for your understanding and support.

I would like to dedicate this book to my parents and my sister. You are always there when I need you, no matter what happens. Your selfless love is the most solid support in my whole life. Yong yuan gan xie ni men!

Finally, to Kaien and Tommie, there is no single word that can express my gratitude to you. You brought countless happiness into my life. I wish I can also be the source of your happiness for the rest of my life.