UNIVERSITY OF OSLO
Department of informatics

Service Discovery in Mobile Ad-hoc Networks

Master thesis

Joakim Flathagen

November 1st 2008

Abstract

Automatic discovery of services and resources is a crucial feature to achieve the expected user-friendliness in Mobile Ad-hoc Networks (MANETs). Due to limited computing power, scarce bandwidth, high mobility and the lack of a central coordinating entity, service discovery in these networks is a challenging task.

In this thesis, I have developed a service discovery protocol (Mercury) utilizing a combination of different optimization techniques: The performance is increased using cross-layer interaction between the application layer and the routing layer. The service information is described using Bloom filters and distributed using Optimized Link State Routing (OLSR). A caching scheme is implemented to obtain further reductions of both overhead and latency.

The analysis and simulation results show that the service discovery proposal induces very low overhead to OLSR and is superior to application-layer solutions. The proposal is implemented as a plugin to the OLSR implementation *olsrd* for real-world deployments.

Preface

This thesis is written as a part of my Master degree in Computer Science at the University of Oslo, Faculty of Mathematics and Natural Sciences, Department of Informatics. The thesis is written at UniK (University Graduate Center) and at Norwegian Defence Research Establishment (FFI).

Acknowledgments

I wish to thank my supervisors Knut Øvsthus, Øivind Kure and Josef Noll for their guidance. I will also thank fellow M.Sc. and Ph.D. students at UniK and University of Oslo. I wish you all good luck in graduating.

Special thanks to my project managers at FFI, Rune Lausund and Lars Erik Olsen, for giving me the possibility to work on this thesis. Additionally, I would like to convey thanks to *all* my colleagues at FFI for your interest and encouraging remarks during the work. Special thanks to Stig Asle Synnes for sharing your math knowledge and Michelle Swearingen for your english lessons.

It would take another thesis to express my thanks to Elin Sundby Boysen.

Joakim Flathagen
November 1st 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If you go out 10 years, computers are not going to be these rectangular objects we carry around. They are going to be extremely tiny. They are going to be everywhere. There is going to be pervasive computing. It is going to be embedded in the environment, in our clothing. It is going to be *self-organizing*.

Ray Kurzweil, 2007

This chapter gives an overview of the motivation behind the thesis and introduces the latest advances in technology that takes part in the research area. The detailed technical background is given in the subsequent two chapters.

1.1 Motivation

The research in this thesis is motivated both by previous research done on mobile computing by the Information and Communication group at the University Graduate Center at Kjeller, Norway (UniK) and by the work done at the Norwegian Defence Research Establishment (FFI). While UniK has been doing research on the latest advances in mobile computing, FFI has been working on providing wearable computing for soldiers. The benefits by equipping every soldier in the battlefield with mobile computers, sensors, navigation equipment and radios are indisputable to increase situational awareness and to reduce fratricide.

However, without a proper design, all the technology could end up being a major logistics– and network administrative challenge and a huge frustration for the individual soldier. It should therefore be of paramount interest to every engineer, researcher and network designer to seek to create systems that prevents errors through an intuitive and user-friendly design.

Soldiers are not the only group that demands portable, robust and networked control systems. The disaster of 9/11, the Hurricane Katrina, and the Asian Tsunami in 2004 have highlighted the need for first responders from different departments and agencies to have common interoperable communication links [63]. Researchers are therefore developing solutions to achieve network connectivity between mobile nodes both in the tactical domain [80], and in the civilian domain [51]. Both areas share the same challenges: High degree of mobility, unpredictable environments and wide range of users—operating in stressed situations.

In such demanding environments, failures are prone to arise. Errors in the original design often leads to human errors—which in turn leads to technical errors and communication failures [64]. One should strive to create a design that hides complexity from the user and therefore reduces the number of failures [68]. A better design also facilitates training [99].

From the network perspective, the term *better design* means robust protocols that performs background processing to automate trivial tasks in order to let the user concentrate on his/her main objective.

To create a well-designed network for demanding customers—such as firefighters, policemen and soldiers—I envision an element of *auto-configuration* to be inevitable. User-friendliness through auto-configuration is in fact the main motivation behind the work in this thesis. A toolbox of standards, products and ideas has recently emerged to facilitate this task. The rest of this chapter introduces some of them and presents the subject of the thesis.

1.2 Overview

1.2.1 Mobile devices

During the last few years, the world of personal computing has seen a paradigm shift. Technology that was until recently available only by the military or in research labs is now a common part of our everyday life. Mobile devices are gaining popularity both in business and for leisure and users can access a myriad of information on demand. A tourist visiting a foreign city can easily check the email, perform a video conference, download electronic maps and browse for the closest sushi restaurant using a mere handheld device. The change-over is happening thanks to all the research done in the terms of microelectronics, wireless devices and software technology during the last decades.

However, not only humans connect. The trend is towards increasingly interconnected networks where electric radiators, vehicles, traffic lights, burglar alarms, biometric monitors, vending machines and lots of other small devices in the environment communicate in *pervasive computing*.

While we are moving towards a world where more and more devices and people are interconnected, we observe that in the same time—as more people embrace the technology—the average user tend to be less sophisticated. More important, the user is *less concerned* about the inner functions of the network technology and *more interested* in using the system as a tool [68]. Again—the need for *auto-configurable* systems and protocols seem inevitable.

1.2.2 Mobile networks

One step in the process to achieve auto-configurable systems is to make the devices able to dynamically form networks—without the use of any preexisting infrastructure such as fixed antennas, access points and repeaters.

Mobile Ad-hoc Network (MANET) technology is a useful tool both to *establish* the networks–

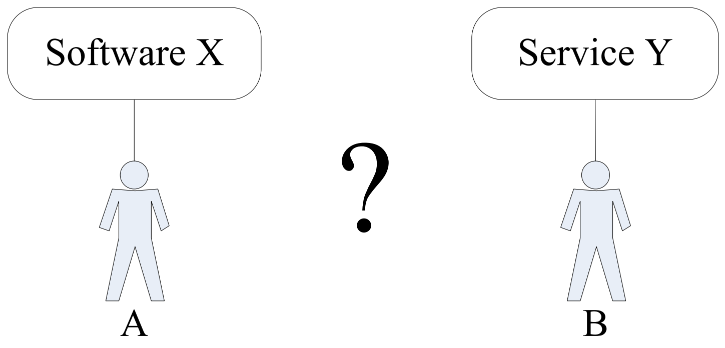


Figure 1.1: Service discovery let the devices find applications and services in the network automatically—without and user-intervention.

without any infrastructure or system administrator—and to *enable communication* between any pair of nodes in those networks. In order to facilitate those two functions, a special routing protocol is employed. The purpose of the routing protocol is to discover rapid changes of the topology in such a way that intermediate nodes can act as routers to forward packets on behalf of the communicating pair.

Early ad-hoc research was mainly aimed at military networks. But, now this technology is attractive to a wide area of applications: Search and rescue operations, vehicle to vehicle networks, tactical networks, virtual classrooms, entertainment and sensor networks are all areas where great benefit can be achieved using the flexibility, ease of maintenance, auto-configuration, and the cost advantages of MANETs.

1.2.3 Services

A mobile network normally consists of users with different roles, various types of equipment, different applications, a handful of sensors, and some shared resources. A way to hide the apparent complexity from the user is to describe all these elements as *services* which can be shared and accessed automatically regardless of their location and ownership.

A *service discovery* architecture let the devices both discover and take use of services in the network, as well as advertise their own capabilities (Figure 1.1). This happens without forcing the user to enter IP-addresses, passwords, user names or other attribute values.

The automatic discovery of services and resources is therefore a crucial feature to achieve the expected user-friendliness of mobile ad-hoc networks.

1.3 Problem statement

Both *Ad-hoc Networks* and the task to *discover services* in various networks have been subject to much research. Both areas are associated with several challenges, and the subject of this thesis: *Service Discovery in Ad-hoc Networks* is far from a trivial task.

The research in the thesis seeks to provide service discovery in a wide range of ad-hoc networked applications. However, the research is specially aimed for bandwidth constrained environments—both civilian and tactical.

1.4 Thesis layout

The thesis is organized as follows:

Chapter 2 gives some background information about mobile ad-hoc networks and general service discovery solutions. The chapter also introduces the taxonomy used to classify the different service discovery architectures.

Chapter 3 gives an introduction about some proposed service discovery techniques for mobile ad-hoc networks.

Chapter 4 introduces Mercury—the service discovery protocol proposed and developed in this thesis.

Chapter 5 addresses different evaluation methods used in ad-hoc network analysis.

Chapter 6 describes the implementation of Mercury for the network simulator ns-2.

Chapter 7 describes the implementation of Mercury for real-world usage.

Chapter 8 describes my choice of simulation and validation methods and evaluates a scenario based on real-world traces.

Chapter 9 tests the most prominent features of Mercury by simulation.

Chapter 10 concludes the thesis and suggests future work.

Abbreviations and acronyms are listed on page 114.

Chapter 2

Background

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat.

Albert Einstein, when asked to describe radio.

As the title of the thesis suggests, the contribution of my research is to combine two technologies: *Ad-hoc networks* and *service discovery*. This chapter first presents routing protocols for mobile ad-hoc Networks. Then, different generic service discovery protocols are presented.

2.1 Mobile Ad-hoc Networks

A Mobile Ad-hoc Network (MANET) is a collection of mobile nodes connected by wireless links able to dynamically form an autonomous multi-hop radio network—without the use of any pre-existing infrastructure. Intermediate nodes in a MANET can act as routers to forward packets on behalf of other nodes. With their self-forming nature and their ability to cope with rapid changes of the topology, ad-hoc networks are attractive to a variety of applications.

However, it is worth noting that ad-hoc networking introduces a great many challenges and imperatives, and also adopts the side effects of wireless computing [17]. Wireless links are significantly less reliable than wired media, they have unpredictable signal quality and transmission range, the channel can be time-varying and possible asymmetric, and the wireless link suffer from security problems not found in wired networks. Further, the multhop nature in MANETs introduces challenges due to the topology dynamics, heterogeneity, variations of node availability and power constrains.

This puts tough requirements to the chosen MANET *routing protocol*. Traditional routing protocols designed for fixed networks such as RIP [36] and OSPF [66] are in general not suited for the ad-hoc environment. The dynamic topology, limited bandwidth and power constraints in MANETs require tailor made solutions. Mainly two different routing approaches are considered in mobile ad-hoc networks: *Reactive routing* and *proactive routing*.

2.1.1 Reactive routing

Protocols in this category are *reactive* in the sense that they only attempt to discover routes between nodes on-demand. Using such an approach, one can lower the total overhead using the protocol in cost of the initial delay finding the optimal route. Reactive protocols are also named *source initiated* or *on-demand* routing protocols. Some examples of such protocols are AODV [75], DSR [45], TORA [73] and DYMO [13]. AODV will be described as an example of one of the most prominent protocols in the category.

AODV

The Ad-hoc On-demand Distance Vector protocol (AODV) aims to obtain routes on-demand, i.e when an upper layer communication packet is destinated to a node not known in the routing table. AODV uses three control messages to obtain and maintain routes:

Route Request (RREQ) A source broadcasts RREQ messages to the MANET if the routing entry is empty for the given destination. AODV can utilize an *expanding ring* technique with gradually increasing Time To Live (TTL) for each request to avoid broadcast storm in the MANET.

Route Reply (RREP) A node replies to a request by sending RREP message either if: (i) it is the destination; or (ii) if it is an intermediate node and has a fresh route to the destination. If the destination is not known, the intermediate node will rebroadcast the RREQ. When a node re-broadcasts a Route Request, it sets up a reverse path pointing toward the source. This reverse path is used to forward Route Reply (RREP) unicast back to the source.

Route Error (RERR) If a node is unable to forward packet, it generates a RERR message. When the originator node receives the RERR, it initiates a new route discovery for the given route.

In addition, AODV performs route maintenance on active routes. If one node in an active path discovers a link breakage, a route error message will be transmitted upstream. The source node will then initiate a new route request.

2.1.2 Proactive routing

In contrast to reactive routing protocols, proactive routing protocols seek to maintain routes to all nodes regardless of upper layer communication demands. By exchanging control messages periodically, the routing table can be kept updated and fresh routes can be provided immediately. Compared to reactive routing protocols, this approach yields more control message overhead, but no initial delay to set up a route prior to communication.

Examples of proactive (or table-driven) routing protocols are FSR [29], OLSR [20], TBRPF [70] and WOSPF [3]. OLSR will be described as an example.

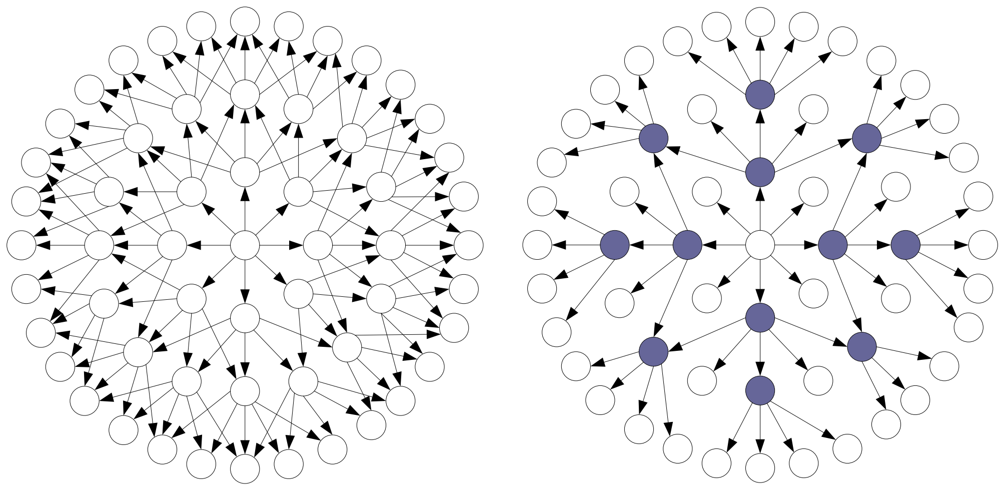

(a) Normal flooding (b) MPR flooding

Figure 2.1: Flooding in a multihop network. Flooding through multipoint relays (MPRs) reduce the number of duplicate transmissions.

OLSR

The Optimized Link State Routing Protocol (OLSR) for MANET is a proactive, link-state routing protocol where each node maintains topology information by periodically exchanging link-state messages. The novelty of OLSR is to employ multipoint relays (MPRs) to minimize the number of control messages flooding in the network. Each node chooses a subset of its one-hop neighbors (MPRs) in such a way that these MPRs will cover all two-hop away neighbors. Hence, messages are only flooded through MPRs, and not to all nodes (Figure 2.1).

Core functioning of OLSR is: Packet format and forwarding; link sensing with hello messages; neighbor detection; MPR selection and MPR signaling; topology control message diffusion; route table computation; node configuration. Three control messages are defined to provide this functionality.

HELLO HELLO messages are exchanged between neighbors only, and diffuse information about the one-hop neighbors of a node. Upon reception of HELLO messages, the two hop neighborhood can be discovered, and further, the MPRs of the given node can be chosen. The MPRs chosen by a node is further marked in the following HELLO messages broadcasted by that node.

TC - Topology Control In OLSR, all nodes chosen as MPR will transmit TC messages. The TC messages contain the address of the node generating the message, as well as the list of nodes that has chosen the given node as MPR (MPR selectors). TC messages are further flooded using the MPRs, disseminating network topology information to all the nodes in the OLSR network.

MID - Multiple Interface Declaration The MID message is broadcasted by nodes running OLSR on more than one network interface.

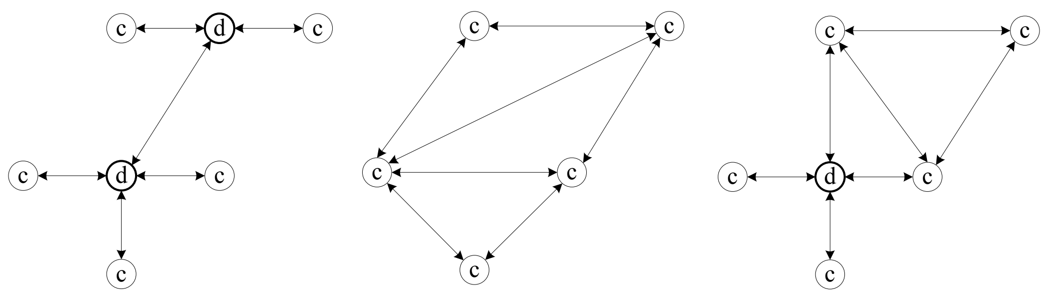

(a) Directory-based (b) Directory-less (c) Hybrid

Figure 2.2: Three different service discovery architectures. Clients (c) either connect directly or via a directory (d).

In addition, a fourth message type, Host and Network Association (HNA) message disseminates information about OLSR nodes that act as gateways (etiher to the Internet or to a separate Ethernet).

Using a common format for all messages the OLSR standard provides extensibility of the protocol without breaking backwards compatibility. This feature gives a unique possibility to disseminate additional information through intermediate nodes even if the nodes do not support the specific extension.

2.2 Service discovery

When the ad-hoc network is established and working, users will obviously want to run different applications. Those application will usually provide or request (or both) services in the ad-hoc network.

In these terms, *service discovery* (or resource discovery) is an important area. Service discovery provides functionality to automatically discover capabilities and to advertise own capabilities to the network. Using service discovery, users can search for services by name, type or class and utilize those services without any further knowledge about the underlying network architecture.

The different service discovery protocols and proposals differ in *architecture design*, *discovery mode* and definition of *service descriptors*.

2.2.1 Architectures

Regarding the dissemination of service information, there are three different architectures available when creating a service discovery protocol.

Directory-based

A directory-based service discovery infrastructure consists of one or several *service directories*. These directories are the only binding between service providers and service clients. Service providers register their services in the directory and clients search for services in this directory (Figure 2.2(a)).

In fixed Ethernet networks, usually *one* node takes the role of the directory. In mobile ad-hoc networks, however, this architecture is not preferable as the directory node will represent a single point of failure and may be out of reach due to mobility. Even within reach, the link to the directory node can be considered unpredictable. Hence, *distributed directories* are preferred. This solution will on the other hand introduce other challenges such as synchronization (between directories, service providers and service clients) and must provide an algorithm to automatically elect new directory nodes if one node fails.

Directory-less

A directory-less architecture omits the use of directories, and use a *distributed* approach only involving clients (Figure 2.2(b)). Hence, there is no need to select directory nodes or to perform synchronization between directories. However, without directories, service requests and advertisements must be disseminated between nodes using either by broadcasting or multicasting. This may induce considerable bandwidth and can be costly in terms of resources on the individual nodes.

Hybrid

Hybrid architectures seek to combine the benefits from the two approaches. Service information is primarily stored on each service provider, but a set of service directories are chosen to be the main binding between services and service requests (Figure 2.2(c)). If a client is aware of an available directory, this directory is preferred. Otherwise, requests are flooded in the network.

Evaluation of architectures

A lot of aspects determine the choice of architecture. The size of the network, the number and type of services, service availability demands and the underlying network protocols are all factors influencing the choice. There is no common consensus on which architecture is the better one. In [81], the hybrid approach is preferred. In contrast, the work in [25] shows that a directory-less architecture performs better than both directory-based and hybrid architectures, partly because the fact that false positive service replies from the service coordinators increase with increasing network dynamics in MANETs.

The hybrid architecture puts an extra load to reactively routed networks, as it triggers additional route requests- and replies compared to the directory-less architecture. This performance issue is obviously not the case in proactively routed networks, where route requests are not on-demand per nature. It should be noted that both directory-based and hybrid architectures introduce complicated mechanisms for electing service coordinators, and put extra load to the infrastructure.

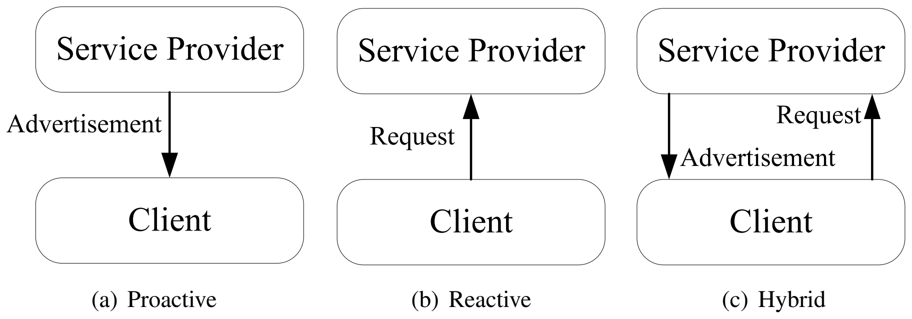


Figure 2.3: Service information can be gathered in different ways.

2.2.2 Discovery Mode

Independent of the chosen service discovery architecture, service information can be gathered either in a *reactive*, *proactive* or *hybrid* way.

Reactive

Using a reactive mode, a service requester node creates a query *on-demand* whenever a certain service is desired (Figure 2.3(b)). The query is then sent to the network either using unicast, broadcast or multicast depending on the service discovery architecture.

Proactive

A proactive mode implies that service providers proactively distributes their available services (Figure 2.3(a)). The distribution is performed either directly to potential service clients or to service directories. Obviously, this approach yields more traffic than the reactive mode. On the other hand, the initial service discovery delay is reduced.

Hybrid

A hybrid discovery mode supports both reactive requests and proactive service advertisements (Figure 2.3(c)). This approach must then support that the service information may be distributed in several ways depending on topology. Some nodes may know all service information, while some nodes have no information at all and must rely on creating service requests.

2.2.3 Descriptor options

There are different approaches to describe the service information in requests and advertisements. Many service discovery protocols use XML to describe the service information. Such a method is adopted in [37]. An other approach is to create service descriptors from ontologies designed for the semantic web services by the use of the Web Ontology Language (OWL) as proposed in [51]. Using such an approach, the ontology must be distributed among the nodes

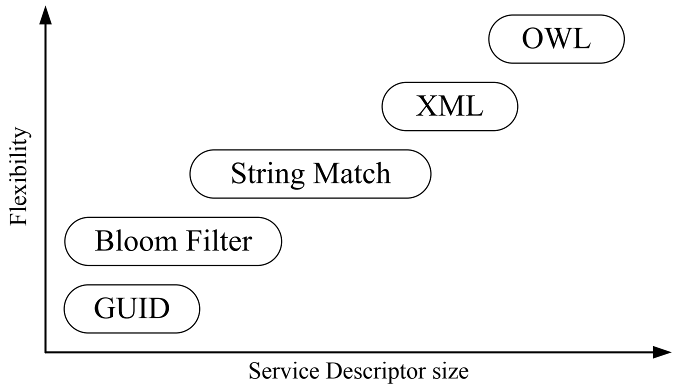


Figure 2.4: Comparing different service descriptor options based on their flexibility in use and their size.

prior to communication. However, both XML and OWL descriptions require considerable bandwidth, which is sparse in ad hoc networks. Some sort of compression could be used to address this deficiency [83] if rich and flexible service descriptors are necessary.

Some proposals seek to reduce bandwidth consumption, and do not see the benefits of using XML or OWL as necessary. By mapping a predefined set of service descriptors, to integers as in [43] or Unique Universal Identifiers (UUIDs) as in [69] the description can be reduced to a few bytes. Such solutions save bandwidth compared to transmitting XML files. However, such solutions are not very flexible nor scalable, as maintenance on every node in the network is required when new service categories are added.

In between those two schools, we find *Bloom filters* [8]. Using Bloom filters, any textual service descriptor can be hashed to a size-defined array without requiring a predefined static set of keywords. In [81], Bloom filters are used to summarize the content of a service directory by hashing the set of WSDL-based service descriptions to a short array.

The different alternatives to service descriptors are compared based on their flexibility (in terms of range of target applications) and their size in Figure 2.4.

2.2.4 Service discovery standards and proposals

The overall Internet community has not yet reached a consensus on one particular service discovery protocol. Several consortiums, companies and organizations have simultaneously been doing research and created their own service discovery protocols. Although most of the proposed protocols do not fit in the ad-hoc environment, a short introduction of the most popular solutions for service discovery will be given. It should be noted that all service discovery proposals made for MANETs—as described in the next chapter—are to a certain extent inspired by the following solutions.

Anycast

A simple way to provide service discovery is to take use of IP-anycast [74]. Using anycast, a client transmits a datagram to a well-known anycast IP address. The routing protocol is then responsible for transmitting this datagram to at least one of the servers that accept datagrams with this address. Using standard routing, the closest server will always be chosen. This functionality simplifies the task of finding a certain server when the user does not particularly care which server is used—like mirrored ftp-servers or DNS-servers.

Although anycast is usable to discover service directories as described in [97], anycast has got several limitations making it difficult to provide a complete service discovery system: First, it is impossible to browse for *all* nodes in a network providing a certain service class, since the routing protocol will only provide an entry to the closest server matching the anycast address.

Anycast is also limited by the fact that there must be provided *one* anycast address for *each* service class in the network. Further, in fine-grained service environments anycast is not the preferred solution, as the protocol does not allow a search for special services. Finally, if a new node enters the network without knowing the anycast address of a service class, no services will be discovered.

Service Location Protocol (SLP)

Service Location Protocol [33] is developed by IETF as a vendor independent standard. The SLP architecture is based on three components: (i) User agents (UA) - which are the software entities that perform the service discovery; (ii) Service agents (SA) - which advertise the location of services; (iii) Directory agents (DA) - which act as central repositories and collects service information from service agents and responds to service requests from user agents. Services and their location are represented as service URLs. UAs and SAs discover the presence of a DA by sending service requests for the DA at startup. The DA also periodically advertises its presence using multicast.

The Service Location Protocol is not widely supported, mainly because dominant companies such as Apple and Microsoft are developing and supporting other service discovery protocols.

Simple Service Discovery Protocol (SSDP)

Simple Service Discovery Protocol (SSDP) [30] is a part of UPnP. UPnP also takes advantage of automatic link-local address choosing [14] to give a auto-configured IP solution. UPnP is included in Windows XP, Vista and several brands of network equipment.

The protocol is based on the following three components: (i) SSDP service—which represents the service agent; (ii) SSDP client—which is the user agent utilizing the services; (iii) SSDP proxy—which is the directory agent representing the binding between the SSDP service and SSPD client. SSDP utilizes unicast HTTP to communicate wit the SSDP proxy. However, the SSDP proxy is not a mandatory part of SSDP, meaning that service information can disseminate in the network without this central entity using HTTP multicast.

Due to the use of HTTP, SSDP is unsuitable for most bandwidth constrained environments.

DNS Service Discovery (DNS-SD)

DNS Service Discovery (DNS-SD) [15] is a way of using the existing DNS records to locate services. DNS-SD was originally proposed by Apple as a part of *Bonjour* (formerly Rendevouz) and also consists of link-local address choosing [14] and Multicast DNS (mDNS). Bonjour can be considered as Apples counterpart to UPnP that is provided by Microsoft.

Bonjour is included in MAC OS X and is used by Apple software such as iPhoto, iChat and iTunes and also supported by the KDE and Gnome desktop environments found on Linux and BSD platforms. Since Apple first launched Bonjour in 2002, every major maker of network printers has adopted Bonjour and uses DNS-SD to advertise the printer service to the local area network [16].

DNS Service Discovery itself is a way of using the existing DNS records to locate services. The protocol can be used to obtain names, service types, port numbers and other attribute information. Since a Bonjour implementation most likely will have a multicast DNS responder for the name-to-address translation, service discovery can be implemented in quite a lightweight manner using the multicast DNS responder to disseminate service information. Even if DNS-SD is considered simpler than SSDP—because it uses DNS rather than HTTP—it is not suitable for low bandwidth ad-hoc networks.

Jini

Jini [87] is a product from Sun Microsystems and is heavily based on Java and Java RMI. In addition to service discovery, Jini provides service invocation, transactions and event notification. Jini allows clients to join a Jini lookup service (JLS), which correspond to the directory agent in the SLP protocol. Using the JLS, the clients can request information about services as well as publish their own services. Publishing a service is performed by uploading a service object to the JLS. This object contains the Java programming interface for the service including necessary methods and applications. The lookup service hence stores Java code necessary for the clients to access the particular service. Discovery is conducted by multicasting a request for a lookup service in the local network.

The bandwidth consumption generated by the discovery process and the fact that Jini is tied to Java and requires a Java Virtual Machine, makes it unsuitable for most low powered embedded systems, including MANETs.

Bluetooth Service Discovery Protocol (SDP)

The bluetooth communication stack contains the Bluetooth Service Discovery Protocol (SDP). The protocol addresses service discovery especially for bluetooth networks.

When searching for services, an SDP client creates a service request PDU containing a search pattern. The search pattern supports searching for services by *name*, searching by *attributes*, and browsing the network for *any* service. A SDP server will respond with a service PDU containing service records for the matching services. The respond can contain additional information as attributes, or the SDP client may request these attributes using a separate request PDU.

2.2.5 Evaluation

Even if the service discovery protocols described in this chapter share much resemblance, they also have different salient features. It should be noted that the protocols are aimed for fixed local area networks or short range wireless networks and are not directly applicable for mobile ad-hoc networks. However, they have served as an inspirational base for developing improvements or completely new protocols tailor made for MANET.

Anycast is a very simple approach and can hardly be called a service discovery *protocol*. Nevertheless, it is a technique suitable for some applications e.g. when searching for a specific server. Developing an anycast routing protocol for ad-hoc networks has proven to be difficult [94]. However, research is on-going to solve this issue in OLSR-based MANETs [23].

Service Location Protocol is although relatively simple, not suitable for MANETs due to its extensive use of directory servers. SLPManet [2] is proposed as an optimization of SLP for MANETs. SLPManet works without directory servers, and also introduces caching in order to reduce the overhead induced in the discovery process. This protocol is further described in the next chapter.

DNS-SD and *SSDP* share much resemblance. Both protocols require an underlying multicast routing protocol—which is not yet standardized for mobile ad-hoc networks. A proposal to optimize SSDP to better suit MANET without using multicast is described in [82]. However, the proposal is prone to generate broadcast storms in the network and is not optimal in terms of bandwidth consumption. Neither DNS-SD is considered usable for MANET without severe optimizations [38].

Jini is tied to the Java programming language, which may not be adaptable for all mobile devices. In networks with no fixed infrastructure—such as mobile ad hoc networks, a reliable connection to the lookup service can not be ensured, making the Jini architecture unsuitable for those networks according to [32]. In [7] Jini is used in an ad-hoc network by taking advantage of a series of adjustments to the protocol.

SDP is a scaled down solution and is only supported on Bluetooth devices, and can therefore not be used in mobile ad-hoc networks. However, as the ideas are simplistic they are applicable for other service discovery solutions tailor made for MANET.

The next chapter covers service discovery protocols aimed for pure mobile ad-hoc environments.

Chapter 3

Related research

Two laptop computers sit less than two feet away from each other. They are so close they are nearly touching—and yet, until recently, as far as network communication is concerned, they may as well have been thousand miles apart.

Stuart Cheshire[1]

The previous chapter introduced general service discovery protocols aimed for local area networks and short-range wireless networks. This chapter describes some of the most prominent service discovery proposals for mobile ad-hoc networks.

3.1 Introduction

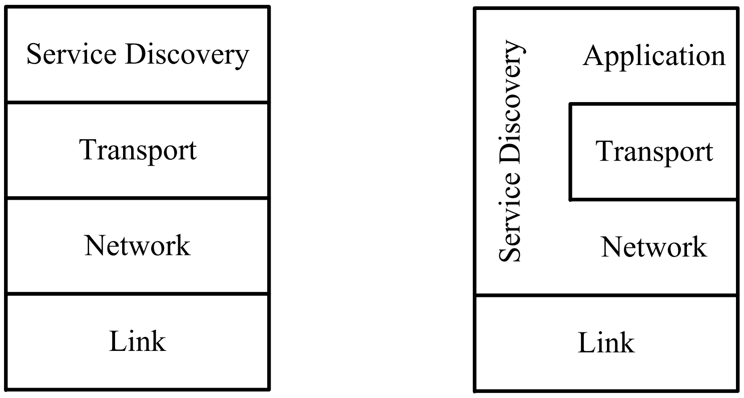
Most of the existing service discovery protocols are primarily designed for fixed networks and are not directly applicable for MANETs without adaptations. Tailor-made solutions specific for MANETs are therefore chosen in favor of more generic solutions. However, since different MANETs vary in size, equipment, applications and objectives, a variety of proposed service discovery architectures for MANET exist to solve specific purposes. Some of the solutions focus mainly at scalability in order to support hundreds or even thousands of nodes. Some solutions seek to minimize latency in the discovery process, while others are focused on reducing the control message overhead to support bandwidth-constrained environments.

Irrespective of the service discovery architecture (directory-based, directory-less or hybrid), or the discovery mode (reactive, proactive or hybrid) there are two possible approaches when designing a MANET service discovery protocol:

Application-layer service discovery Refers to protocols *independent* of the underlying routing protocol

Cross-Layer service discovery Refers to protocols *integrated* with the routing protocol, be it either *reactive* or *proactive*.

[1]from the book ”Zero Configuration Networking” [16].



(a) Application-layer service discovery (b) Cross-layer service discovery

Figure 3.1: Service discovery protocols for MANETs work either at the application layer or are considered as cross-layer protocols.

Most MANET service discovery proposals belong to the first category, and place service discovery at a layer *above* routing (Figure 3.1(a)). Such mechanisms create an overlay on top of the network layer to disseminate service advertisements, requests and replies in the network. There are several advantages using this method: (i) as no assumption is made about the underlying network, it is possible to create pervasive service discovery architecture across different networks domains. (ii) The architecture can be based on existing standards, since the size of the service descriptors is not limited by the routing protocol. (iii) A modular and layered approach is maintained making it possible to replace protocols at any layer.

Cross-layer service discovery is motivated by the need to optimize control overhead and reduce service-acquisition latency. As both the service process and the routing process must coexist in an ad-hoc network—both processes generate and receive messages. It is therefore possible to *exploit* the routing layer for efficient dissemination of service control messages (Figure 3.1(b)).

Different service discovery proposals from both categories will now be described.

3.2 Application-layer service discovery

Most application-layer service discovery protocols are ambiguous in the terms that on the one hand they strongly support the layered approach and claim to be *independent* of the underlying network architecture. On the other hand, they *rely* on network-layer support to multicast or broadcast the service discovery messages. If the target MANET supports multicast, application-layer service discovery leads to a simple and modular design. Thus, it is possible to disseminate service discovery through intermediate nodes that does not run any service discovery code (Figure 3.2). However, it should be noted that multicast in MANETs is still at the research stage (no standard is defined) and is hence an open issue.

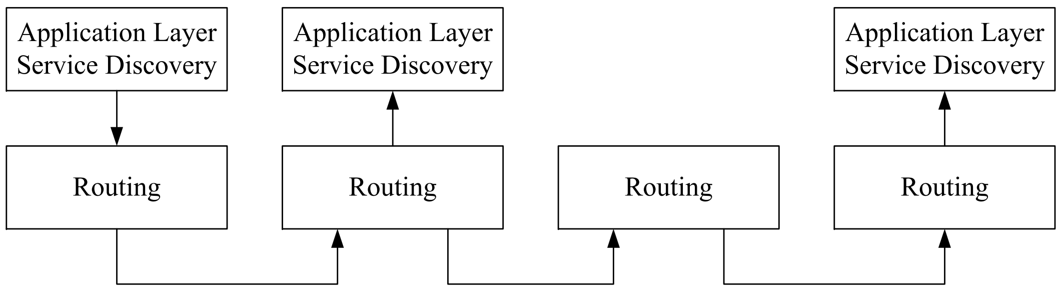

Application Layer Service Discovery
Application Layer Service Discovery
Application Layer Service Discovery
Routing
Routing
Routing
Routing

Figure 3.2: Using service discovery at the application level, all nodes in the network can forward service discovery messages.

3.2.1 Pervasive Discovery Protocol

The Pervasive Discovery Protocol (PDP) is a *directory-less* protocol aimed for ad-hoc networks [12]. Each PDP node has a User Agent (PDP-UA) and a Service Agent (PDP-SA). The PDP-UA-process search information in the network and the PDP-SA process advertise services offered by the device. For each advertisement, an availability time is included. Entries are removed from the cache of each node when the availability timer runs out without being updated.

PDP operates in reactive mode and assumes that the underlying network is either a one-hop network, a multi-hop ad-hoc network with multicast routing support.

3.2.2 Konark

Konark [37] is a *directory-less* service discovery architecture based on a peer-to-peer model using lightweight HTTP servers. The protocol defines its own description language loosely based on WSDL (Web Services Description Language) [59]. HTTP and SOAP [55] are utilized to handle service delivery. The Konark architecture maintains a tree-based structure to cope with service classification. The format support search for either *all*, *generic* or *specific* services in each category and the requests can be done either using simple keywords or a more fine-grained service description if a specific service is desired.

Using this classification, a node can search for *a general printer*, or choose to do a more detailed search for *a color laser printer on the second floor*. Konark supports both proactive and reactive service advertisements, and both servers and clients can actively discover and advertise services on a need basis. A service request is sent to a fixed multicast group, and all the nodes with a matching service will respond. A time-to-live field is specified in the service advertisement process and enables local caching of service descriptors on each node.

3.2.3 SLPManet

SLPManet [2] is an adaptation of Service Location Protocol [33] to make it work in MANET environments. The most prominent change from SLP is that Directory Agents (DAs) are omitted from the protocol, making the architecture *directory-less*. As a consequence, Service Agents only reply on requests from User Agents (reactive mode), in contrast to SLP, which supports both proactive and reactive discovery between Service Agents and Directory Agents. Optional SLP messages such as Attribute Request and Service Type Request as defined in the RFC are not implemented in the MANET adaptation.

The protocol includes a simple caching scheme, where all nodes in the network cache service information for a certain time. Caching increases performance but—as anticipated in [2]—cache entries may be false when the network topology changes. This is a general problem in all application-layer designs without access to routing information (i.e. non cross-layer designs).

3.2.4 Sailhan

Sailhan et.al has proposed a *directory-based* service discovery architecture[2] aiming at large-scaled ad hoc networks [81].

Directories in Sailhan are distributed and deployed dynamically. The directories form a virtual backbone of nodes exchanging service requests and replies using WSDL service descriptors. The architecture is by definition independent of the routing protocol, and communication between directories is done using a special bordercasting technique. The bordercasting is inspired by MPR flooding used in the OLSR routing protocol [20]. The architecture can also take advantage of the OLSR MPR election itself instead of creating the bordercast overlay on its own. Hence, even if the service information is distributed at the application layer, the protocol can utilize some cross-layer optimizations.

The dynamically allocated directory agents are deployed so as at least one directory is reachable in at most a fixed number of hops. Directories then cache the descriptions of services available in their vicinity and uses Bloom filters [8] to summarize the content of the directory by hashing the set of WSDL-based service descriptions.

3.3 Cross-Layer service discovery

Cross-layer design refers to protocol design done by actively exploiting the dependence between protocol layers to obtain performance gains [85]. By doing this, cross-layer solutions may violate the modular layered approach. A violation of a layered architecture involves giving up the luxury of designing protocols at different layers independently. Such optimizations should therefore be used with caution as cross-layer interactions can have undesirable consequences on system performance [47].

However, in some situations, cross-layer interactions are inevitable to eliminate the redundancies associated with repeating similar tasks found on adjacent layers [86].

Cross-layering in a service discovery context means *all* optimizations done by taking advantage on information found on lower layers—such as examining the routing table or measuring signal quality. Henceforth, the term *cross-layer service discovery* refers to service discovery solutions that utilizes the routing process to disseminate service discovery messages. Routing-layer support was first introduced by Koodli and Perkins [50]. Now, several different proposals exist both for reactively routed and proactively routed MANETs.

[2]In this thesis, I take the liberty to name the architecture *Sailhan*.

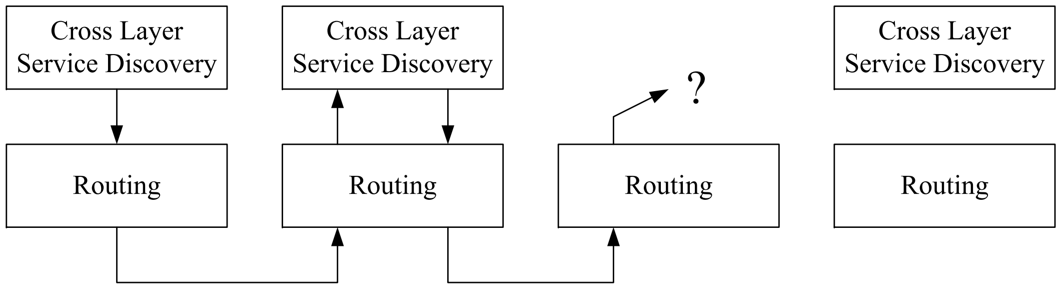

Cross Layer Service Discovery
Cross Layer Service Discovery
?
Cross Layer Service Discovery
Routing
Routing
Routing
Routing

Figure 3.3: Some cross-layer service discovery proposals require changes to the routing protocol. Intermediate nodes without those modifications prevent successful service discovery.

3.3.1 Reactively Routed MANETs

SEDRIAN is a *directory-less* service discovery architecture relying on AODV as routing protocol [69]. Service information can be described in two different ways: An optimized description using a 128-bit Universal Unique Identifier (UUID) for generic services, and a more descriptive language to advertise special services that cannot be described in a simple UUID.

SEDRIAN exploits AODV by encapsulating three new packets in the AODV RREP message: DREQ (Discovery Request) contains a request for a UUID-based service. DREP (Discovery Reply) contains a reply to a discovery request. The last message is ADVM (Advertisement Message). It contains the advertisement any special service provided.

It should be noted that since SEDRIAN uses the AODV RREP message to disseminate discovery requests, replies, and advertisements, the proposal brings severe changes to the original AODV protocol. The RREP is not originally used as a broadcast message in AODV, and some adjustments are therefore necessary to avoid packet loops. For this reason, all nodes in the network must support SEDRIAN in order to make the discovery process work. Any AODV node in the network without the SEDRIAN extension, will prevent service discovery messages to be disseminated (Figure 3.3).

The proposal by Engelstad et.al [26] bears resemblance to SEDRIAN, but utilizes AODV in a slightly different manner. In addition, the protocol can be implemented both *discovery-less* and as a *hybrid* architecture.

Using a discovery-less architecture, service discovery requests (SREQ) are piggybacked on AODV Route Request Packets (RREQ), and service discovery replies (SREP) are piggybacked on AODV Route Reply packets (RREP). Using the hybrid architecture, the service coordinator announcements are piggybacked on AODV RREQ packets and service registrations are piggybacked on AODV RREQ packets. Using this technique, there is no need to change the original AODV code, except allowing piggybacking of service discovery messages.

A thorough study of service discovery in reactively routed MANETs can be found in [98].

Protocol	Service descriptor	Dissemination	Routing	Architecture	Mode
PDP	Text	Multicast	Any	Directory-less	Reactive
Konark	WSDL	Multicast	Any	Directory-less	Hybrid
SLPManet	SLP-url	Multicast	Any	Directory-less	Reactive
Sailhan	WSDL	Multicast	Any	Directory-based	Hybrid
SEDRIAN	UUID+	Cross-layer	AODV	Directory-less	Reactive
Engelstad	Not defined	Cross-layer	AODV	Hybrid	Reactive
Jodra	Fixed integer	Cross-layer	OLSR	Directory-less	Proactive
LSD	Not defined	Cross-layer	OLSR	Hybrid	Hybrid

Table 3.1: Comparison of different service discovery proposals

3.3.2 Proactively Routed MANETs

Jodra et.al [43] present a solution on integrating service discovery with the OLSR routing protocol. The different OLSR messages [20] share a common message header. Utilizing this header, a new message called Service Discovery Message (SDM) is introduced. The SDM packet can contain either a service advertisement or a query.

The proposal also introduces a service cache for each node in the network. The cache stores all services available, both local and foreign. Whenever a node wants to use a service not stored in its local cache, it sends a request asking for the service using the SDM query message. SDM messages are forwarded by the MPRs in the network. Upon receiving a SDM query message, a node checks whether its local services corresponds to the service asked for in the query SDM. If this is the case, it will send an advertisement message announcing the requested service. The answer is MPR flooded.

As the complete SDM message is only 8 bytes, and thanks to piggybacking of the SDM to the OLSR packets, only a small overhead is added to the network. However, it should be noted that this efficient, albeit limited message format, cannot cope with advanced and detailed service descriptors. Further, the service descriptors must be a priori known to all the nodes running service discovery.

A similar proposal, which also utilizes OLSR as routing protocol, is Lightweight Service Discovery (LSD) [57]. A message similar to the previously mentioned SDM, Service Location Extention (SLE), is here introduced. LSD supports both *directory-less* and *directory-based* architectures. Using the latter architecture, the discovery mode can be both reactive and proactive. The proposal bears resemblance both to [43] and to ideas presented in [24].

3.4 Summary

This chapter has described a variety of different service discovery proposals for MANETs with different salient features. The features are summarized in table 3.1.

Most proposed service discovery solutions solves service discovery at the application layer, arguing that cross-layer solutions violates a modular layered approach and hinder easy interchange of routing protocols. However, a cross-layer integration of the service discovery architecture with the routing protocol seems to bring considerable optimizations and the benefits are

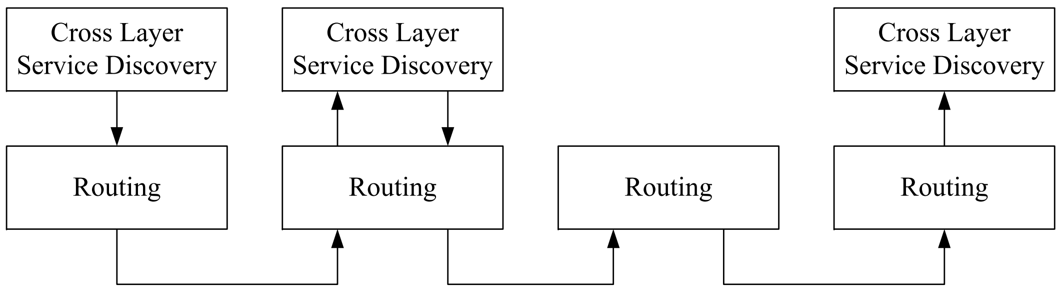

Cross Layer Service Discovery
Cross Layer Service Discovery
Cross Layer Service Discovery
Routing
Routing
Routing
Routing

Figure 3.4: The OLSR default forwarding algorithm forwards service discovery packets without having to change the original OLSR code.

indisputable both in proactively and reactively routed MANETs. As the focus in this thesis is on low-bandwidth environments, I state that cross-layer solutions are inevitable.

It is important to use a service discovery architecture that is *transparent* to avoid that every node in the network must run service discovery code. *Unaware* nodes in the network should be able to forward requests, replies and advertisements on behalf of other nodes. Using an application-layer design with multicast dissemination, such a transparent architecture is possible (3.2). But, as shown in Figure 3.3, not all cross-layer proposals support such a transparent concept and require changes to the routing protocol to ensure packet forwarding of service discovery messages.

The choice of routing protocol is a matter of operational scenario, traffic patterns, available bandwidth, delay requirements as well as a matter of personal taste. Research favoring both reactive, and proactive protocols can be found. I envision a cross-layer service discovery protocol based on the following consepts:

- It should use OLSR to aim for transparency.[3] The OLSR default forwarding algorithm forwards all packets (also new and unknown packet types) using MPR forwarding (Figure 3.4).
- The architecture should be directory-less to eliminate the overhead with selecting and maintaining directories.
- It should include caching to reduce control traffic and to lower the discovery delay.
- Service descriptors must be defined in an efficient and flexible manner.
- It should be tailor made for low-bandwidth environments and seek to lower the overhead on the routing protocol.

The rest of this thesis describes and evaluates a new service discovery protocol that seeks to fulfill the requirements above.

[3]An OLSR-network such as Freifunk Berlin [28] consists of over 700 nodes. Any pair of nodes in this network can take immediately use of an OLSR-based service discovery protocol without having to update any code at the other nodes.

Chapter 4

Mercury - A cross-layer service discovery protocol

Why can‘t computers in real life work like they do on *Star Trek*? (...) They don‘t have to do all that careful handwork involving cables and IP addresses and logins and passwords on *Star Trek*—it all just works. Is it just special effects, or are *we* missing something?

Paul Vixie

Considering the requirements drawn in the previous chapter, I propose a new service discovery protocol. This chapter provides a design overview of the new protocol.

4.1 The design

As different MANETs vary both in size, equipment, applications and objectives, a variety of different protocols to solve service discovery in these networks are proposed and implemented. Some of them have been described in the introduction of this thesis.

The aim in this thesis is to create and evaluate a new service discovery design for *low-bandwidth* ad-hoc networks primarily aimed at tactical and first responder networks. In order to make an efficient service discovery solution suitable for such environments, I envision several optimization elements to be included. These elements are:

- Service information (service descriptors) must be defined in an *efficient* manner and should be *scalable* to support several simultaneous requests or advertisements.
- The service discovery process must rely on efficient service descriptor dissemination to save bandwidth.
- The solution should be fully *distributed* both to optimize for speed and to obtain redundancy.

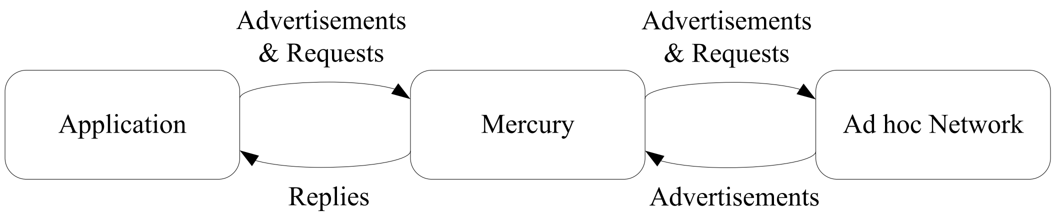


Figure 4.1: Mercury connects users and applications to services in the ad-hoc network using service advertisements and service requests.

In this thesis I propose a new service discovery protocol— subsequently named Mercury[1]. The protocol works in the following manner: The service descriptors are described using *Bloom filters*. The service dissemination is done by *piggybacking* service information on OLSR routing messages, and the solution is fully distributed and utilizes intelligent *local caching* with service handover support.

The purpose of Mercury is to act as a common framework that connects services distributed in the ad-hoc network to users and applications (Figure 4.1). I will now describe the different components of Mercury.

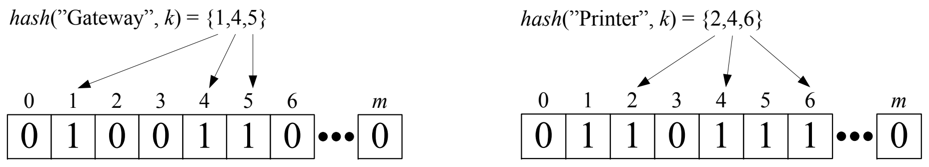
4.2 Service description by Bloom filters

The proposed solution in this thesis is to distribute a summary of the available services as a vector (or array) described as a *Bloom filter* [8]. The technique of Bloom filters was originally used primary in database applications, but due to the interesting characteristics of Bloom filters, the technique has received attention in many aspects of computer network research. Both peer-to-peer networks, packet routing, data measurement, dictionary systems and password checking are applications that can benefit from Bloom filters [10]. Bloom filters can be considered in any application where implementation space of a list is important and a small amount of *false positives* can be accepted.

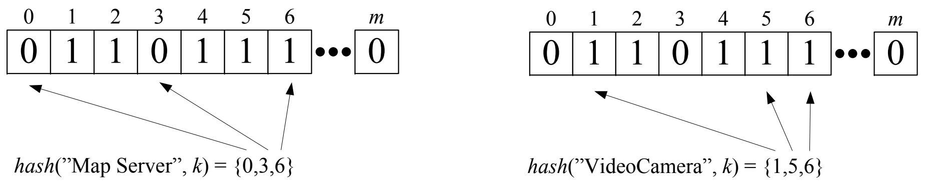
For our purpose, a Bloom filter is an efficient way to describe services. Using such filters, it is possible to summarize all available services on one particular node or directory in a small size-defined array. The approach gives network efficient and timely service dissemination.

In dense networks with a high number of available services, there is a chance for *false positive* service replies: A node may falsely respond positive to a query even if the requested service is not actually offered. The false positive rate can be minimized by analyzing the filter and then setting the different parameters of the filter to optimal values.

[1]In roman mythology, Mercury was the messenger of the gods and the major god of trade, profit and commerce. Light footed, and with winged sandals, he carried urgent messages for the other gods.



(a) The first service is added to the Bloom filter (b) The second service is added to the Bloom filter



(c) This service is not found in the filter (d) This query yields a false positive

Figure 4.2: A Bloom filter of m bits is used to store service descriptors. Two services are added to the filter. A query of the service "Map server" shows correctly that this service is not part of the filter. However, the Bloom filter responds positive to the query "VideoCamera" even if the service is not actually part of the filter (i.e. false positive).

4.2.1 Introduction

In our context, the intention of the Bloom filter is to represent a set $S = \{x_1, x_2, \ldots, x_n\}$ of n service descriptors in an efficient manner. We start by defining a Bloom filter v implemented as an array of m bits. All the bits $\{1, \ldots, m\}$ are initially set to 0. The filter uses k independent hash functions $h_1, h_2, \ldots, h_k$ with range $\{1, \ldots, m\}$ to hash each service descriptor x to the array v.

For each service descriptor $x \in S$, the hash output $h_i(x)$ represents an array position in v, $v[h_i(x)]$ that is set to 1 for all hash functions $i = 1, 2, \ldots, k$. One location in v can be set to 1 multiple times, however it is obvious that only the first change has any effect. Figure 4.2(a) and 4.2(b) illustrates two different services hashed by three hash functions and then added to the same array (or filter).

In order to check whether any service z is in the Bloom filter, we have to determine whether all $h_i(z)$ are set to 1. If this is the case, we assume that the service z is avaliable. If all $h_i(z)$ are not 1, the service is not part of the filter—as in Figure 4.2(c). The Bloom filter may, however, yield a false positive if the filter indicates that a service descriptor $z \in S$ even though it is not (Figure 4.2(d)). The chance of getting a false positive lookup can be estimated using calculus of probability.

4.2.2 False positive calculation

Given that m is the length (in bits) of the Bloom filter, n is the number of service descriptors inserted in the filter, and k is the number of hash functions used, the false positive probability is given by equation 4.1. Calculations can be found in Appendix A.1.

$$P_{fp} = \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{4.1}$$

Notice that the number of services is the only value that can vary while the application is running It is therefore important to have a thorough understanding of the target application and to set the parameters k and m carefully to minimize the probability of false positive queries.

There are two ways to reduce the chance of false positives: One approach is to change the number of hash functions k. The second method is to increase the size of the Bloom filter itself, namely m.

The optimal number of hash functions

The optimal value of k can be calculated by taking the derivate of the equation 4.1 (See Appendix A.2 for calculations). We then find that the optimal number of hash functions, k_{opt}, for a filter of width m and a certain number of service descriptors n is:

$$k = \frac{m}{n} \ln 2 \quad \Rightarrow \quad k_{opt} = \lfloor k \rceil \tag{4.2}$$

When designing a service discovery protocol based on Bloom filters, equation 4.2 is important in order to choose the best number of hash functions. The number is given by the expected number of services to be stored and the filter width reasonable with respect to the transmission protocol and radio medium limitations. In Mercury, the default number of hash functions is four.

The size of the filter

In order to minimize the false positive rate, the filter (m) should mathematically be as large as possible—preferably indefinite. However, computation time, network data rate and memory consumption limits the feasible size of the filter.

Figure A.1 in appendix A.3 shows how the false positive vary by changing the number of hash functions and the size of the filter. In Mercury, the default filter size is 128 bits.

4.2.3 The Mercury Bloom filter

The heart of the Bloom filter is the hash function. Any hash function can be used as long as it is able to map an item (e.g. service descriptor) to a pseudo-random number uniform over the range $1 \ldots m$. Equally important: the outcome of the k different hash functions must be independent. One way to implement a hash function is to use a series of modulo functions (such as in SBDM hash). Another approach is to use a cryptographic hash function such as MD5 [77].

Even if MD5 is considered insecure for most cryptographic purposes today, it has desirable properties as a basis for a Bloom filter hash function. MD5 is deterministic and uniform, and has excellent collision resistance for our purpose. MD5 also exist as open source code for many programming languages, and implementations are relatively fast. Due to its qualities, the false positive probability can be brought close to the theoretical limit, given by equation 4.1.

It should be noted that most cryptographic hash functions—even MD5—are due to its tampering resistance, computationally slower than general-purpose hash functions. However, the MD5 process is run only upon advertising and requesting a service and not when service matching is performed—as matching is done on the filters itself. Further, as shown subsequently, only *one* MD5 operation is required to generate input to all k different hash functions.

The Mercury Service discovery design uses MD5 in the following manner: The k hash functions, which constitutes the Bloom filter, are constructed from k groups of each r bits out of the 128 bit hash from the MD5 operation. Any set of sub-bits from an MD5 output can be used as an input to an independent function. Each of these k functions sets one bit in the filter v.

In Mercury, the hash functions are implemented as shown in Algorithm 1.

Algorithm 1 Calculate Bloom filter value v for service x

```
Require: $x \neq 0$
1: $a \Leftarrow MD5(x)$
2: $r \Leftarrow 128/k$
3: for $i = 0$ to $k$ do
4:     $f \Leftarrow subbits(r * i, (r * (i + 1)) - 1, a)$
5:     $v[f \mod m] = 1$
6: end for
```

A thorough evaluation of MD5 as Bloom filter is described in appendix A.4.

4.2.4 Summary

The benefits of using Bloom filters to describe services can be summarized as follows:

- The filter provides an optimized description in terms of number of bits.
- Several service descriptions can be transmitted simultaneously without increasing the array size.
- Any textual service descriptor—independent of its size—can be added to the filter. This gives an enormous flexibility.
- As the services are not distributed in a human-legible manner, only the applications that know the name of the service can utilize it. Hence, ”hidden” resources and services can be advertised in the network.

With a thorough understanding of the target application, and by correct parameter settings of the filter, it is feasible to create a Bloom filter based service discovery protocol that is superior to clear text service descriptor dissemination. By example: A number of 10 arbitrary service descriptors can be advertised in a 8 byte filter with less than 5% probability of a false positive. Text or XML-based services require 10-100 times the space.

4.3 Protocol Format

Bloom filters are used to describe both advertised and requested services. The Mercury service discovery protocol is used to distribute the filters using a protocol format that extends the OLSR routing protocol.

The default control messages employed in OLSR are HELLO and TC and are communicated using a unified packet format [20]. Each OLSR message transmission can consist of several such messages *piggybacked* to the main header (Figure 4.3). OLSR also gives an unique possibility to disseminate different kinds of information through intermediate nodes even if the nodes do not support the specific message. Unfamiliar messages will still be forwarded using the default-forwarding algorithm (Figure 3.4 on page 21).

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Packet Length         |    Packet Sequence Number     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Message Type |     Vtime     |         Message Size          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Originator Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time To Live |   Hop Count   |    Message Sequence Number    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
:                            MESSAGE                            :
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Message Type |     Vtime     |         Message Size          |
|+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Originator Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time To Live |   Hop Count   |    Message Sequence Number    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
:                            MESSAGE                            :
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
```

Figure 4.3: OLSR packet format, from [20]

Each message in the OLSR transmission has an individual header, which permits special treatment. The originator can for example limit the flood by a diameter in terms of number of hops or add a certain validity time for the message.

Mercury service discovery integrates with the extensibility feature of the OLSR standard by introducing the Mercury service discovery message (MSD) (Figure 4.4). MSD messages are sent as the data-portion of the general message format with the message type set to MSD_MESSAGE. The Time To Live field is set to 255, but can be used to perform an expanding ring search for services in a future version.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      | Filter Length |             Spare             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
:                         Service Filter                        :
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.4: Mercury service discovery format

The Mercury service discovery message consists of four fields (Figure 4.4):

Type This field specifies that the message carries one out of two message types:

> **SD_REQUEST** This indicates that the message is a service discovery request. The service filter field consists of one or more services ($\{1 \dots n\}$) that the sender node is requesting.
>
> **SD_ADVERTISEMENT** This indicates that the message is a service discovery advertisement (or service reply). The service filter field consists of *all* the services ($\{0 \dots n\}$) that the sender node is offering.

Filter Length This field gives the size of the service filter (m), counted in bytes. The filter is limited upwards to 2040 bits.

Spare This field is not used and should be set to 0 by the originator of the message[2].

Service filter This field contains the filter describing the messages to be requested or advertised. The filter is encoded as a Bloom filter. The size (in bytes) is given by the field "Filter Length".

The filter length is limited upwards to 2040 bits, which is sufficient for most practical purposes[3]. It is important to limit the total length of the service discovery message in order to prevent interruption of regular OLSR control message transmission and to facilitate piggybacking of several messages to one single header. The number of messages that can be simultaneously piggybacked to one common OLSR message header is limited by the individual size of the messages and the Maximum Transmission Unit (MTU) of the underlying medium.

4.4 The discovery scheme

Mercury handles requests and advertisements from two entities (Figure 4.1 on page 24): (i) Local applications on the node, and (ii) foreign nodes through the ad-hoc network. Each node

[2]In the ns-2 extension described in chapter 6, the spare field is used to store a message sequence number for service requests and service replies in order to facilitate statistics.

[3]Referring to calculations and discussions in appendix A.

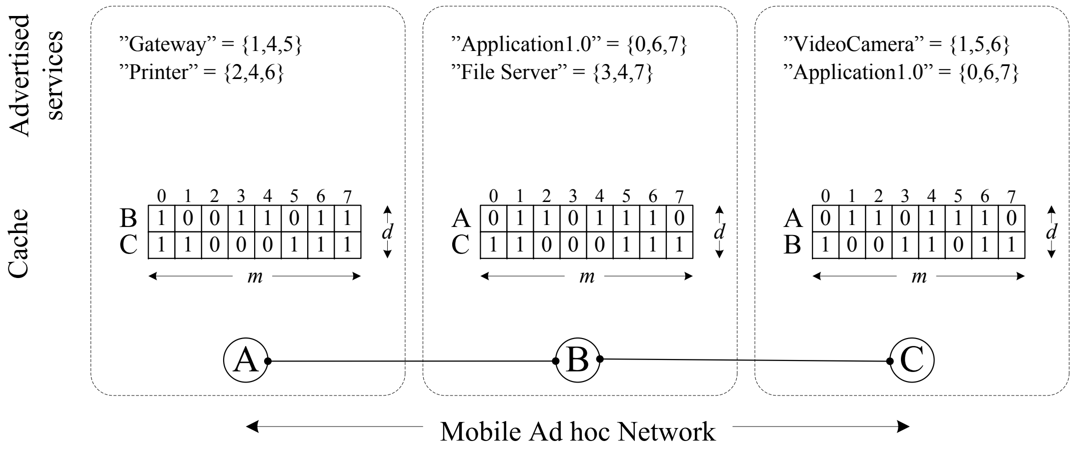


Figure 4.5: Each node in the Mobile Ad hoc Network employ two repositories: One repository store the local services advertised, and one repository—implemented as a attenuated Bloom filter of depth d—serves as a cache storing advertisements received from foreign nodes.

uses a set of repositories to store the information (Fig. 4.5): **Advertised services** contains the different services offered by the node itself. In Foreign services **cache**, all the services offered by other nodes are stored. Each entry in the list consists of the Bloom Filter advertised by the foreign node and its current IP address.

The last repository contains the **Requested services** which stores all the services requested—awaiting an incoming advertisement.

4.4.1 Sending service advertisements

All applications that locally connect to Mercury service discovery *advertise* their own services. Textual service descriptors from an application are immediately added to a Bloom filter that contains *all* the services offered by the given node. Subsequently, an SD_ADVERTISEMENT message is created containing this summary Bloom filter and flooded through the network using MPR flooding.

4.4.2 Sending service requests

A service request from an upper layer application is immediately queried in the local cache to check whether any of the foreign nodes has previously advertised the particular service. If no match in the local cache is found, an SD_REQUEST message is created as a Bloom filter containing *all* the services the given node is requesting and sent using the MPR flooding technique.

4.4.3 Receiving service advertisements

Each node in the ad-hoc network employs a cache to store incoming service advertisements. The cache is implemented as an *attenuated Bloom filter*[4]. For our purpose, the attenuated Bloom filter, which constitutes the cache is an array of depth d containing d normal Bloom filters. Each filter contains the services advertised by one single foreign node.

An advertisement received from a network node is immediately added to (or updates an existing entry in) the local cache. In addition, incoming service descriptors are immediately matched against the recent service requests in the repository containing the requested services. If a match is found, the local application is notified about the successful service discovery.

4.4.4 Receiving service requests

A request received from an external node immediately queries the local cache. If a matching service is found, the node replies with a service advertisement. The service advertisement is created as a Bloom filter containing *all* the services the node is currently advertising, and sent using the MPR flooding technique.

4.5 Additional features

4.5.1 Path-aware caching

The protocol uses local caching of services advertised by foreign nodes in order to save network bandwidth and reduce the discovery latency. Local caching may, however, lead to false positive replies to the overlying application if a service exists in cache even if the node that advertised the service is not available anymore.[5] The cache timeout is therefore a trade-off between fast service queries and the false positive rate.

To solve this issue, Mercury includes a simple addition to the discovery process that consults the local routing table for the availability of the node offering the service. The technique is outlined in Algorithm 2. Without this *path-aware* construction, false positive replies as a side effect of caching are more likely to happen. Such false positive replies cause unacceptable delays and reduce user satisfaction since the upper layer application has to time out on the false request.

[4]Attenuated Bloom filters were introduced in [76] for use in a probabilistic routing algorithm.

[5]Solutions relying on service directories suffer from the same problem.

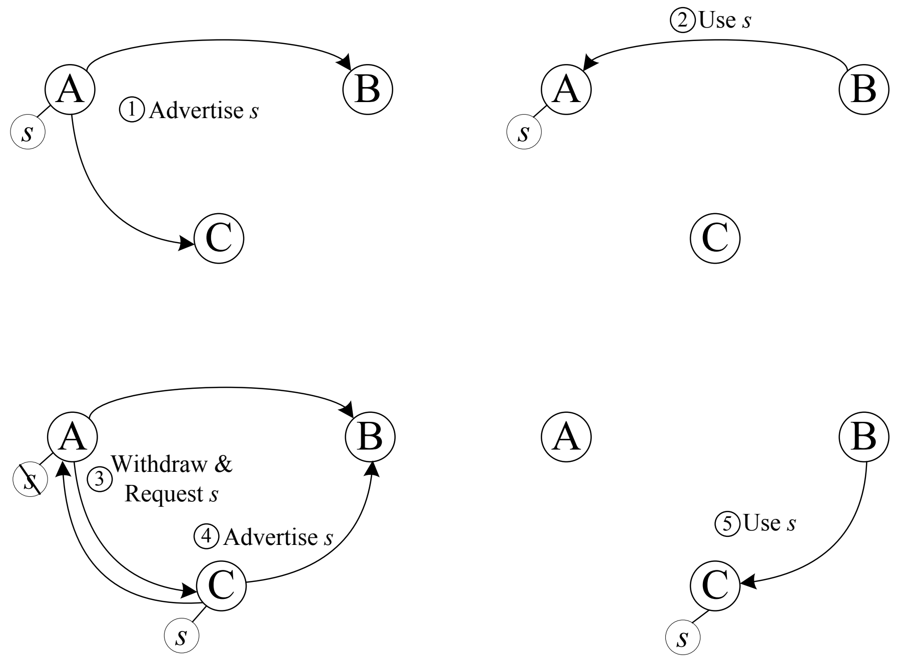


Figure 4.6: A service s is advertised by A and used by node B. When A withdraws the service, it immediately requests the service in order to speed up the discovery process for other nodes (here: B). Luckily, node C then advertises a similar service s which B can put to immediate use.

Algorithm 2 Returns the address of the node offering the service x

```
1: if $LookupLocalCache(x) > 0$ then
2:     $Node \leftarrow LookupLocalCache(x)$
3:     if $ExistInRoutingTable(Node)$ then
4:         return $Node$
5:     end if
6: end if
7: $Node \leftarrow PerformServiceDiscovery(x)$
8: return $Node$
```

4.5.2 Service handover

In a fully distributed service oriented network, the *service availability* is a compound of several factors. A node dependent of a certain service may fail to reach the service due to mobility, network congestion or user failures. A failure may also arise if the node offering the service shuts down the application or resource offered while it is being utilized by a foreign node or user. If the latter incident happens, the service discovery system is responsible for providing a service of the same service class from another source (if available) to the affected node in a timely manner.

The Mercury service discovery protocol supports such a *service handover* function initiated when an application withdraws a service.

When an application shuts down or for some reason cannot provide a particular service (say s) anymore, it withdraws the service to inform other nodes immediately that the service is about to become unavailable. The withdrawal process consists of the following stages:

1. When a service s is locally withdrawn from node A, s is immediately removed from the advertised services repository. A new service advertisement message is then created as a Bloom filter containing all the *remaining services* the node is offering. Notice that even if this message implicitly contains special information—as the number of advertised services just decreased (the number may even be zero)—it is a regular service advertisement message.

2. Prior to sending this special case advertisement message, node A does a neat operation: Since any other node may be dependent on s (like node B on figure 4.6), node A creates a *service request message* containing service s—even if service s is not of interest for A per se.

3. Subsequently after creation, the two messages (the service advertisement message and the service request message) are sent using MPR flooding (step 3 in figure 4.6). The two messages will be piggybacked[6] to the same OLSR message header.

4. Any node providing a service of type s will immediately respond to the service request (step 4 in figure 4.6). This respond will be received by all nodes in the network.

5. Nodes dependent of service s will then contact the new service provider immediately (step 5), and the service is effectively handed over.

If the service is not explicit withdrawn using this technique, the following scenario is likely to happen (with respect to figure 4.6):

1. An application on node B will fail to reach the service s since it is removed from node A, and the application will eventually time out the connection after time T_{APP}.

2. When the time T_{APP} has elapsed, the application will initiate a new service discovery requesting s. If the local cache entry of s (pointing to A) has not yet timed out (as it will automatically after time T_{CACHE}), the service discovery protocol will reply with a false positive telling the application that service s still exists on A even if it does not. A false positive may also occur if the node A itself is not available anymore. On the other hand, if path-aware caching is implemented (Algorithm 2 on the preceding page), a new service discovery is initiated, which will result in a new entry pointing s to C.

Consequently, after withdrawal of s, an application may have to wait as long as $T_{APP} + T_{CACHE}$ to regain the service even if a similar service has been available the whole time. This delay may stop the application from performing its tasks and therefore vastly reduce the user satisfaction. With the withdrawal scheme outlined above, combined with path aware caching, a seamless handover is possible.

[6]Whether they are piggybacked or send separately is dependent on the OLSR queue status and jitter settings.

4.6 Summary

The protocol described in this chapter introduces Bloom filters as an efficient way to describe arbitrary service descriptors. The protocol uses efficient service descriptor dissemination using MPR flooding. Further, the protocol employs local caching to lower the discovery latency. Cross-layer techniques are used to minimize the false positive probability when caching is used. The architecture is fully distributed and supports both proactive and reactive discovery.

The remaining part of this thesis will describe the implementation and evaluation of the protocol for two different purposes: First, the protocol is implemented as an extension to a network simulator. Second, the protocol is implemented as a plugin to OLSR for real-world experiments.

Chapter 5

Evaluation methods

> Practical wisdom is only to be learned in the school of experience. Precepts and instruction are useful as far as they go, but, without the discipline of real life, they remain the nature of theory only.
>
> Samuel Smiles

This chapter addresses the different evaluation techniques used in ad-hoc network analysis. First, widely used evaluation techniques for contemporary ad-hoc network research are discussed. Then, the techniques used in this thesis to evaluate Mercury service discovery are introduced.

5.1 Introduction

When designing protocols and algorithms for mobile ad-hoc networks, a major part of the research is concentrated on evaluating and analyzing the design in order to prove that the approach is sound and hopefully outperforms existing comparable solutions.

Four well-known techniques exist to help evaluation and analysis of ad-hoc network protocols and algorithms:

- Analytical modeling
- Simulation
- Emulation
- Real-world experiments

Each of the techniques above has its own set of advantages and disadvantages which will now be discussed briefly.

5.1.1 Analytical modeling

Mathematicians often use analytical models to evaluate certain protocol properties. Analytical evaluation is inevitable when analyzing the performance of IEEE 802.11 [6], and to examine probabilistic mobility models [11]. Analytical modeling is also a valuable tool when evaluating data structures such as Bloom filters (analyzed in [8] and in chapter 4 in this thesis).

Results from analytical modeling are both precise, resource effective and often portable to adjacent problems. However, analytical modeling may be hard to understand for fellow researchers without in-depth knowledge of the field. Therefore, to make analytical models practicable and possible to interpret and adjust, they are usually limited to one single algorithm or part of a protocol. Hence, it is rarely feasible to make an analytical model to deal with entire protocol stacks.

5.1.2 Simulation

Simulations is an established and widely used method to conduct performance evaluation of network components [48]. Simulators such as ns-2 [92], GloMoSim [91] or OMNeT++ [72] come with built-in support for the most popular network protocols; e.g. IP, TCP, UDP, Ethernet and Wi-Fi. The entire protocol stack can thus be simulated at once, enabling validation of new protocols or algorithms implemented as additional code or scripts.

The major benefits by performing simulations are the repeatability (other researchers may verify the results) and scalability (simulations can cope with hundreds or even thousands of nodes).

Certain approximations and simplifications are, however, often made when simulation models are used, which can lead to biased conclusions. Simulations of ad-hoc networks have for example been criticized for not using valid mobility models [96] or by relying on *one* specific scenario [53]. The network simulator itself may also include errors or assumptions such as unrealistic wireless medium characteristics [52, 60].

However, despite the many pitfalls and possible errors when performing simulations, performance evaluation by simulation is virtually inevitable in order to validate the scaling characteristic of a protocol design.

5.1.3 Emulation

Using emulation, hardware- and software components originally designed for real-world deployment are combined with simulation components. The purpose of emulation is often to test protocols and algorithms on real hardware preparing for real-world experiments. The emulator can work on the physical layer, the MAC layer, or at combinations of different protocol layers. By changing various parameters such as antenna attenuation and signal propagation, node movement can be effectively emulated.

Emulations can be done in special test beds as surveyed in [52]. Different emulation test beds are used to evaluate distinct features of different ad-hoc network components. Even some network simulators, such as ns-2 [92], can be used as a limited-functionality emulator.

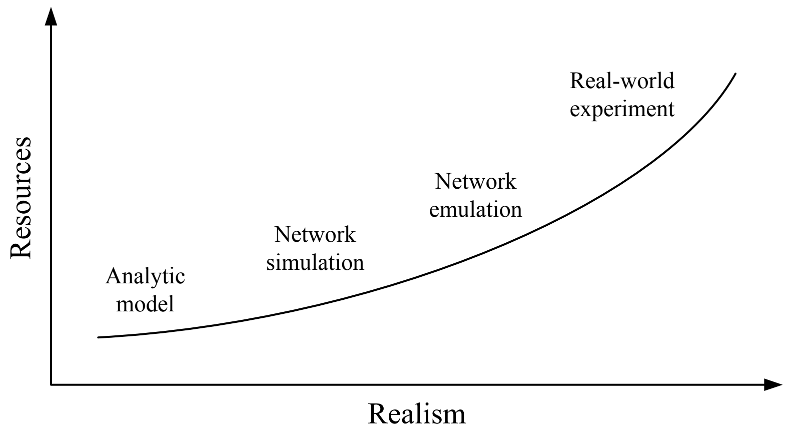


Figure 5.1: The resources, such as time money and human hours, necessary to perform validation of a protocol or algorithm increases when a high degree of realism is necessary.

Emulation is a valuable tool that is considered a compromise between simulation and real-world experiments regarding cost, time and repeatability. If software code can be reused, emulation is a valuable tool to accelerate the switch to real-world experiments.

5.1.4 Real-world experiments

Real-world experiments have recently gained popularity among researchers to validate simulations or analytical models [52]. Real implementations of ad-hoc and mesh networks such as [56, 28] are paramount to prove that algorithms and protocols work as expected when deployed in an operational network. In a real-world setting, all components and parts of the system are fully functional (albeit using an experimental environment). A recent survey of real-world implementations [48] concludes that protocols and algorithms *must* be evaluated in real-world settings in order to address all aspects of the design.

The drawback by using real-world experiments is obviously the time, money and human resources required to perform the experiments. Further, results from experiments are often non-reproducible and hard to validate. It is also difficult to isolate and test one particular behavior of the investigated protocol.

5.2 Summary of evaluation methods

Even if the credibility of simulations is a subject of discussion [53], most research in the field of mobile networking today purely rely on simulations to evaluate the characteristics of a protocol design. This choice is not unfounded: Time, money and human resources increases when moving from simulation to emulation, and increases vastly when moving to real-world experiments, as illustrated by Figure 5.1.

Taking into consideration the different benefits by choosing different evaluation methods, researchers should aim to evaluate new designs by creating a test-bed including several methods.

A new protocol design can for example be evaluated both by a number of initial simulations and then by implementation and test in a real-world scenario.

5.3 Evaluation techniques for service discovery

Evaluation of a particular service discovery design can be performed using any of the described methods—assumed that the researcher takes the different benefits and drawbacks into consideration. The choice of method also depends on resources, knowledge, hardware availability, the service discovery architecture (e.g. which protocol layer it belongs to) and personal preference.

Some service discovery proposals are only evaluated analytically or by an architectural description [69, 97]. Others are purely evaluated by simulations [2, 43, 12, 57, 81] while others are implemented for small real-world experiments [34, 37, 38].

Different service discovery proposals often take advantage of completely different scenarios in their simulation and experiments. Hence, studies are difficult to compare. The individual choice of scenario is, however, not unfounded, as different service discovery solutions are targeted to completely different applications—and *one single* evaluation approach is most certainly not valid for all solutions.

In order to reflect the variety of configurations found in different MANET implementations, Abou et al. provide a set of benchmark scenarios to evaluate different service discovery proposals [1]. Although the evaluation scenarios vary to cover a wide range of uses, all the scenarios rely on simulations as the one and only validation method and do not consider emulation or real-world testing.

5.4 Evaluation of Mercury

In this thesis, a combined approach is taken. The Mercury service discovery protocol is implemented *both* for the network simulator ns-2 *and* as a plug-in to OLSR. These two implementations make both simulations and real-world experiments possible. The next two chapters describe the implementation details of the two different approaches respectively.

Chapter 6

Implementation for ns-2

To write good software you must simultaneously keep two opposing ideas in your head. You have to be able to think *how hard can it be?* with one half of your brain while thinking *it will never work* with the other.

Paul Graham

In order to verify and evaluate the Mercury service discovery protocol in a variety of scenarios, network simulations are inevitable. This chapter describes the implementation of the service discovery protocol integrated with the ns-2 network simulator. Notice that chapter 7 covers the Mercury implementation for real-world usage. These two implementations are, although different, similar in many respects. Duplicated information is therefore avoided to a certain degree, and both chapters should be read in order to get a full understanding of the Mercury service discovery protocol.

6.1 The ns-2 network simulator

6.1.1 Introduction

Many discrete-event network simulators are suitable for MANET research. Among the most popular simulators we find GloMoSim [91], OMNeT++ [72] and J-Sim [88]. In this thesis I choose to use the ns-2 network simulator [92].

The ns-2 simulator can be downloaded from the Internet, *free of charge*. Hence ns-2 is often preferred rather than expensive commercial alternatives. The simulator also works on *different platforms*. Although ns-2 is far from platform independent, all that is needed to make the simulator run is (in theory) a mere computer equipped with a C++ compiler.

Further, the ns-2 simulator is the *most popular* network simulator today. According to a survey performed by Kurkowski et al. [53], ns-2 is by far the most used simulator in MANET research: 43.8% of the papers studied in the survey used ns-2, 10% used GloMoSim, and the rest of the papers used either a less popular commercial simulator or a self developed simulator.

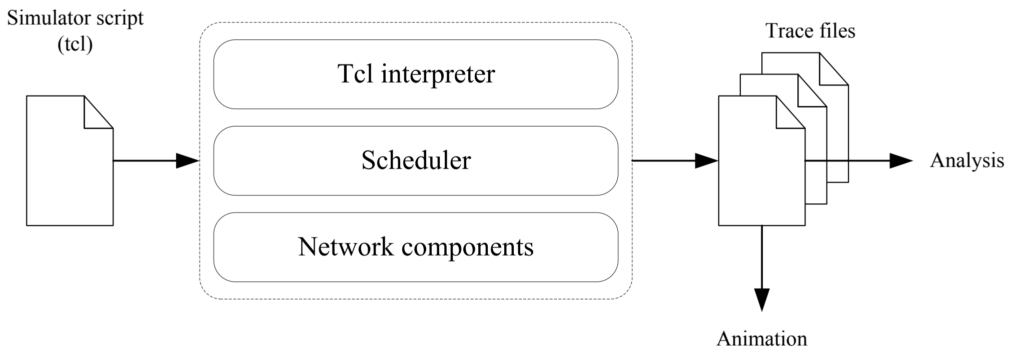


Figure 6.1: The main components of the ns-2 network simulator.

There are several benefits by using the most popular network simulator: (i) Bugs in the simulator are more likely to be discovered and corrected. This is extremely important, as it is paramount that the output from the simulations is valid and can be trusted. (ii) It is more likely to find MANET researchers familiar with ns-2 than any other network simulator. Most researchers can therefore verify simulations performed by others. (ii) It is feasible to base a study or part of a study on research accomplished by other researchers since much work and studies are available.

On the downside, the ns-2 simulator has a steep learning curve and is therefore often considered hard to use. And due to its command line interface, ns-2 may seem bothersome for researchers familiar with point-and-click graphical user interfaces.

6.1.2 Functionality

Simulations in ns-2 are user-defined by tcl-scripts. The simulator interprets the scripts at simulation startup and performs the simulations via a scheduler. The scheduler uses both built-in and auxiliary network components. The Mercury code described in this chapter is an example of such an auxiliary component. The simulations result in one or several text files (Figure 6.1). Those text files can be interpreted by an external program to collect statistics, or be used as input to an animator program[1] in order to visualize the node mobility and the packet flow.

6.2 UM-OLSR

6.2.1 Introduction

The network simulator ns-2 comes with several built-in routing protocols such as AODV, TORA and DSR. The OLSR routing protocol—which is a fundamental part of the Mercury service discovery protocol—is, however, not a part of the default ns-2 implementation. A third-party implementation must therefore be included subsequent to the ns-2 installation. In this thesis, I choose to take advantage of UM-OLSR [93] from the University of Murcia. The implementation is extended to include the service discovery messages and algorithms introduced in chapter 4.

[1]The Network Animator (NAM) [92] is the most common visualization tool for ns-2.

In addition to UM-OLSR, there are other available implementations of OLSR such as NR-LOLSR [67] or INRIA [41]. All of them complies with the rfc [20] and support all core functions. I chose UM-OLSR because the code is freely available and fairly easy both to understand and to extend. The readability of existing code is important when a new code extension—such as a cross layer service discovery implementation—have to infiltrate major parts of the existing code.

6.2.2 Functionality

It is not the intention in this thesis to thoroughly describe every function in UM-OLSR. The reader is referred to [93, 78] for details about UM-OLSR not given in this chapter. The most prominent features is, however, explained in order to give an understanding of the tight coupling between the routing protocol and the service discovery implementation.

As any other protocol extension to ns-2, UM-OLSR acts as an agent listening to all packets of a certain packet type flowing in the simulator. When a simulated node is about to receive a packet classified as an OLSR packet[2], the packet is handled by UM-OLSR. UM-OLSR then parses the packet and does appropriate actions as defined in the OLSR standard. UM-OLSR handles its own timers, enabling the automatic transmission of OLSR control packets.

One important feature of UM-OLSR is that most of the OLSR parameters are configurable from TCL simulator scripts. I.e., there is no need to recompile the simulator when changing parameters such as HELLO intervals, TC intervals, willingness etc. This feature is important when comprehensive simulations examining the effect of variable alteration are performed. The Mercury implementation extends this feature.

6.3 Service Discovery Implementation

Due to space limitations (and obviously with respect to the readability) the source code of the entire service discovery implementation is not included in the thesis. Some details about the code structure plus a small simulation example will be explained in order to give an understanding of the implementation.

As illustrated by Figure 6.2 on the following page, the Mercury service discovery implementation acts as a code extension to UM-OLSR. The extension is made both by altering the original UM-OLSR code and by adding new source code files to the structure. Using the new code, UM-OLSR is able to handle auxiliary messages such as service requests and replies as if they were original OLSR messages.

The Mercury service discovery system consist of two main components: (i) A simple protocol format (including Bloom filters) defines the service requests and service advertisements. (ii) Arrays (or repositories) to store various service information.

[2]Internally in ns-2, OLSR packets are classified as PT_OLSR

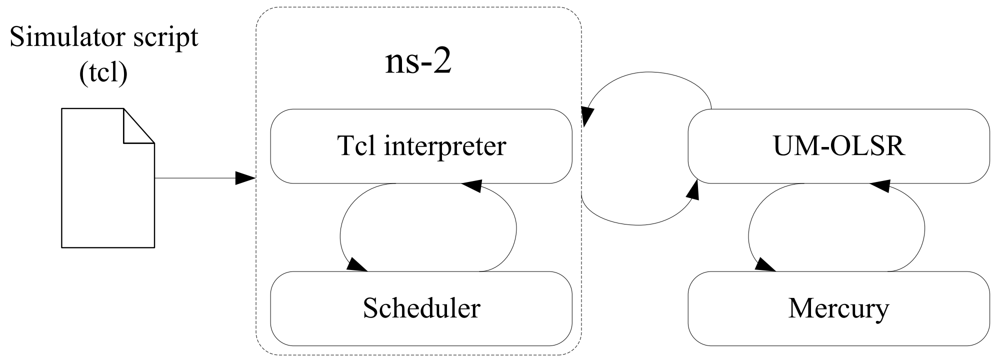


Figure 6.2: The UM-OLSR implementation extends ns-2 with OLSR routing capabilities. Mercury provides cross-layer service discovery by taking advantage of UM-OLSR.

6.3.1 Protocol format

Three building blocks are involved in the transmission of service messages: Original ns-2 code, UM-OLSR, and the Mercury extension. When ns-2 identifies a general OLSR packet, the internal scheduler calls the function `OLSR::recv` in UM-OLSR. If the message is considered valid, the function `OLSR::recv_olsr` is immediately called. The latter function processes each of the OLSR messages within the packet in turn. Each of these messages might be *either* one of the defined OLSR control messages recognized from the RFC [20] (such as HELLO, TC and MID) *or* the Mercury service discovery message as outlined below.

The service discovery format is defined in `olsr_pkt.h` of the UM-OLSR structure along with the basic OLSR messages:

```
typedef struct OLSR_sd {

   u_int8_t type_;
   u_int8_t filter_length_;
   u_int16_t spare_;

   inline u_int8_t& type() { return type_;}
   // The 16 spare bits used as sequence number in ns-2
   inline u_int16_t& seqnum() { return spare_;}

   // Service Filter
   unsigned char bloomFilter[filter_length_];
   inline u_int32_t size() { return filter_length +
                             sizeof(type_) +
                             sizeof(spare_) ;}

} OLSR_sd;
```

As illustrated above, the message format description complies with the format defined in chapter 4 with one exception: The 16 spare bits (which are originally unused) are used as a sequence number. The intention of the sequence number is to be able to trace one particular service message (say advertisement) while it traverses through simulated nodes inside ns-2.

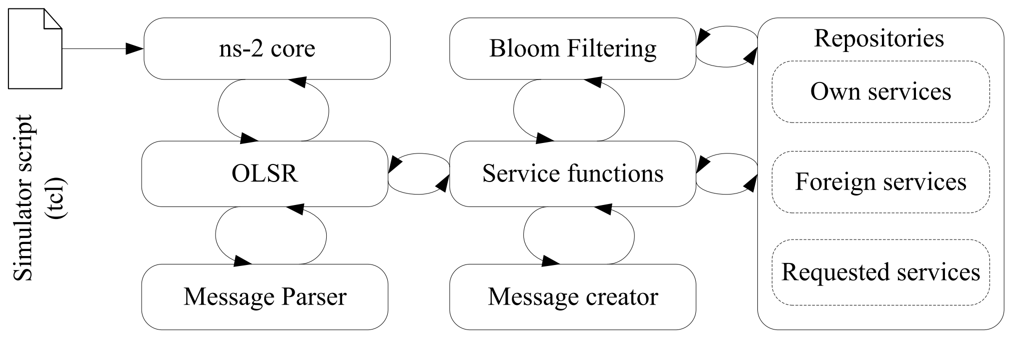


Figure 6.3: Service discovery functionality is made accessible for ns-2 scripts through the OLSR extension. All service information used by the simulated network node is stored in the information repositories.

6.3.2 Repositories

The core of the implementation evolves around the storage of service-data in information repositories (Figure 6.3). The basic functionality of the repositories is earlier described in 4.4 on page 30. The first repository handles services advertised by the local node. The second repository stores services advertised by foreign nodes, and a third repository stores the service requests awaiting reply. All repositories are implemented as a series of C++ Standard Template Library (STL) vectors.

Figure 6.3 illustrates the relation between core ns-2 functionality, OLSR, message parsing, message creation, Bloom filters and repositories. The core of the Mercury service discovery is the service functions, which act as the glue that connects all the different components.

6.3.3 Service functions

The service functions make it possible for the ns-2 script to invoke service discovery in the simulated network. If one of the nodes participating in an ns-2 simulation (say node 7) wishes to advertise the service ”Map-server” at time 0.0, the following command is entered in the tcl-script:

```
$ns_ at 0.0 "[7 agent 255] SD_ADD_SERVICE Map-server"
```

Similarly, if node 6 wishes to request the very same service at time 20.0, the following command is entered:

```
$ns_ at 20.0 "[6 agent 255] SD_REQUEST_SERVICE Map-server"
```

The tcl commands above indicates that implementing a simulation script with Mercury service discovery is rather straightforward. The next section describes a complete (albeit simple) simulation.

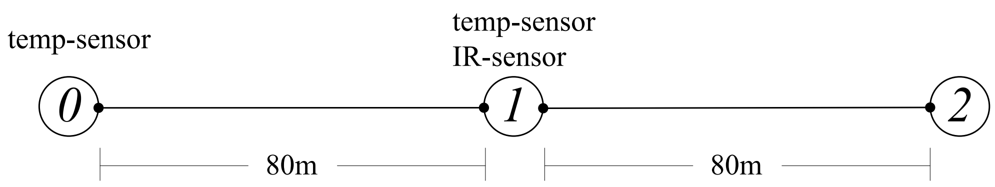


Figure 6.4: A simple OLSR network. Node 0 *and* 1 *offers services, while node* 2 *requests services.*

6.4 Example simulation

The scenario illustrated by Figure 6.4 is used as a basis to demonstrate how a simple service discovery simulation can be created. It is assumed that the reader possesses base knowledge about ns-2 and is familiar with tcl-scripting. Nevertheless, any reader familiar with computer programming should be able to follow the example. The entire example is attached in appendix B on page 97, and the most prominent parts of the example will be described subsequently:

6.4.1 Configuring Mercury

Mercury-enabled network nodes can be configured using simple TCL-commands:

```
Agent/OLSR set sd_proactive_ false
Agent/OLSR set sd_ival_ 10
Agent/OLSR set sd_cache_ 300
Agent/OLSR set sd_numhash_ 4
```

In the first line in the above listing, we define that Mercury will operate *reactively*. The other operation mode is *proactively* behavior. Proactively in this context means that available services will be advertised within intervals given by `sd_ival_`. The default (and preferred) operation mode is, however, the reactive mode. Even if proactive mode may yield lower discovery delays compared to reactive mode, it will certainly lead to increased data traffic. A sort of hybrid mode is enabled by setting a high `sd_ival_` and enable proactive mode.

The third line defines the cache timeout (in seconds). The fourth and last line define the number of hash functions to be used by the Bloom filter algorithm (discussed in 4.2.2 on page 26).

6.4.2 Define topology

The network topology is defined by a simple set of commands defining the X, Y and Z coordinates within the boundaries of the simulated area:

```
$node_(0) set X_ 100.00
$node_(0) set Y_ 100.00
$node_(0) set Z_ 0.0000
$node_(1) set X_ 180.00
$node_(1) set Y_ 100.00
```

```
$node_(1) set Z_ 0.0000
$node_(2) set X_ 260.00
$node_(2) set Y_ 100.00
$node_(2) set Z_ 0.0000
```

The above excerpt defines the location of the three nodes according to the setup illustrated by Figure 6.4. The transmission range is set to 100m (by setting the `RXThresh_`, see appendix B on page 97 for details). Traffic between node 0 and node 2 is OLSR routed through node 1.

6.4.3 Define service access

The service advertisements and requests are defined according to the definition in 6.3.3:

```
$ns_ at 1.0 "[$node_(0) agent 255] SD_ADD_SERVICE temp-sensor"
$ns_ at 2.0 "[$node_(1) agent 255] SD_ADD_SERVICE temp-sensor"
$ns_ at 10.0 "[$node_(1) agent 255] SD_ADD_SERVICE IR-sensor"
$ns_ at 20.0 "[$node_(2) agent 255] SD_REQUEST_SERVICE temp-sensor"
$ns_ at 20.1 "[$node_(2) agent 255] SD_REQUEST_SERVICE IR-sensor"
```

Using the simple commands above, the service ”temp-sensor” is advertised by both node 0 and 1, and the service ”IR-sensor” is advertised by node 1 alone. Node 2 requests both services. The first request occurs at time 20.0 and the second at 20.1.

With most of the simulation script explained, we are now ready to run the simulation.

6.4.4 Running the simulation

Assumed that ns-2 and the UM-OLSR protocol are installed properly (see [92, 93]), and that the Mercury service discovery extension is a part of the source tree, the simulation is run by typing `ns sd.tcl outputfile.tr`. After a successful run, the trace file `outputfile.tr` can be examined.

6.4.5 Examining the trace file

Trace files from ns-2 simulations tend to be quite large. Even the small simulation in this example generates a trace file of 915 lines. Using a simple grep[3], we can isolate the entries containing the interesting service discovery information. Notice that superfluous information irrelevant for our purpose is excluded from the traces for the sake of readability.

Adding service descriptors

The first service discovery related information in the trace file is the service advertisement done by node 0 and 1. We can observe that the services are successfully added to the repositories of each of the nodes.

[3]Referring to the Unix command *grep* which finds text within a file.

```
sd 1.0 0 SD_ADD_SERVICE temp-sensor
sd 2.0 1 SD_ADD_SERVICE temp-sensor
sd 10.0 1 SD_ADD_SERVICE IR-sensor
```

Requesting the first service

At time 20.0, node 2 requests the first service, namely ”temp-service”:

```
sd 20.0 _2_ SD_REQUEST_SERVICE temp-sensor
sd 20.0 _2_ SERVICE_NOTFOUNDINCACHE temp-sensor 0 0 2
```

Notice that since Mercury is running in pure reactive mode, node 2 can not find the service ”temp-sensor” in the local cache (foreign service repository on Figure 6.3). Node 2 therefore immediately sends a service request containing a Bloom filter hash of the service descriptor.

```
s 20.000000000 _2_ RTR  --- 39 OLSR 64 [[SD REQ 0 2 0 12]]
```

The above trace line tells that a service request (`SD REQ`) is sent (marked with an ”s”) as part of an OLSR message. Let us examine what happens next:

```
r 20.001088267 _1_ RTR  --- 39 OLSR 64 [ [SD REQ 0 2 0 12]]
sd 20.0010883 _1_ SERVICEFOUND
```

Observe that the request is received by node 1 (marked with an ”r”), which immediately searches through its own advertised services (Own service repository on Figure 6.3), and then prints out that the service is found. This information is then immediately sent to all network nodes:

```
s 20.001088267 _1_ RTR  --- 40 OLSR 96  [ [SD ADV 0 1 0 15][SD REQ 0 2 1 12]]
```

Notice that the above message contains two service messages piggybacked into one single OLSR message: The first part of the message (`SD ADV`) is the positive respond to the query. The second part is the original query (`SD REQ`) from node 1 which is MPR forwarded. Node 0 will then receive the forwarded message:

```
r 20.002492533 _0_ RTR  --- 40 OLSR 96 [ [SD ADV 0 1 0 15][SD REQ 0 2 1 12]]
sd 20.0024925 _0_ SERVICEFOUND
```

Node 1 will, similarly as node 0, search through its own advertised services and send a positive feedback to the request. Shortly thereafter, node 2 receives the positive reply from node 1:

```
r 20.002492533 _2_ RTR  --- 40 OLSR 96  [ [SD ADV 0 1 0 15][SD REQ 0 2 1 12]]
sd 20.0024925 _2_ SERVICETRUEPOSITIVE temp-sensor 0
```

As we can see from the above two trace lines, the advertisement message from node is now successfully received at node 2, and the service discovery process is completed. Notice that the entire process took place in less than 2.5ms.

Requesting the second service

In the simulation script, we defined that node 2 should initiate a second discovery for the service ”IR-sensor” after 20.1 seconds. The trace file below presents one prominent feature of the Mercury service discovery, namely the caching:

```
sd 20.1 _2_ SD_REQUEST_SERVICE IR-sensor
sd 20.100000000 _2_ SERVICEFOUND_CACHE IR-sensor 1 1 0 1 2
```

As implied by the trace lines above: The first service discovery process did not only yield a successful discovery of the service ”temp-service”. The reply from node 1 also contained *all* the services offered by node 1. Hence, the second service request performed by node 2 at time 20.1 resulted in a mere cache-lookup yielding a discovery time of 0.0s.

6.5 Summary

In this chapter, the implementation of the Mercury service discovery protocol for the network simulator ns-2 is explained, and a simple simulation is described. This implementation is used for all subsequent simulations in this thesis. No solution should, however, exist solely in a simulator, and the work in the next chapter brings the protocol closer towards real-world deployment.

Chapter 7

Implementation for olsrd

Talk is cheap. Show me the code.

Linus Torvalds

This chapter describes the implementation of Mercury Service Discovery protocol aimed for real-life usage. The *olsrd routing daemon* and its plugin interface is introduced, and the Mercury Service Discovery plugin to olsrd is presented.

7.1 Overview

The proposed cross-layer design requires a tight coupling between the service discovery protocol and the OLSR routing protocol. For this reason, there is a need for a complete and well-written OLSR implementation flexible enough to cope with additional code extensions. Different OLSR implementations exist and can be used in such a test and development stage. Most of the implementations are open source and can be downloaded free of charge, such as *nrlolsr* [67] and *OOLSR* [41]. The most popular and well documented implementation today is, however, the UniK olsrd project [71] and the Mercury service discovery implementation will be based on this implementation. I refer to olsrd as the *implementation*, and I use the uppercase abbreviation OLSR to specify the *protocol*.

7.2 The UniK olsrd daemon

The olsrd project was originally based on an open source project from INRIA, but was later heavily modified as a part of a masters thesis [89] at University Graduate Center at Kjeller (UniK). During the project development, the source was redesigned to become fully compliant to RFC 3626 [20] and became available on the Internet [71]. The project is now embraced by

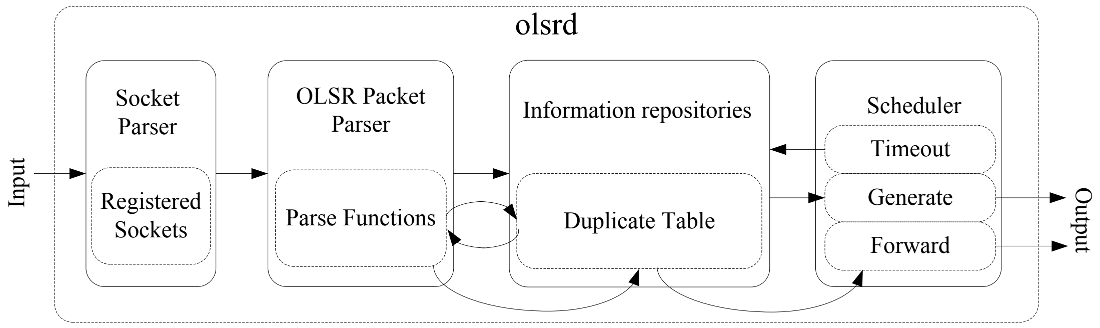


Figure 7.1: OLSRd basic functionality, based on a figure in [89]

the open source community and thanks to a lot of researchers and engineers, olsrd is among the most popular implementations of the OLSR routing protocol.

The olsrd developers are very active and are continuously updating olsrd with new improvements and extensions. The latest version of olsrd now runs on multiple platforms, such as Linux, Windows, OS X, and iPhone. Both source code and binaries can be found on the web site [71] available for free download.

Olsrd is now used in several test beds and real networks such as to create Internet access in rural areas [56] or in cities like Berlin and Rome [28, 95]. Olsrd is also used by Norwegian Defence Research Establishment (FFI) to provide communication in the digitized battlefield [4].

7.2.1 Core functionality

Even if olsrd is rather complex, the core functionality is kept simple and is easy to understand. Figure 7.1 outlines the basic parts of olsrd and their relation.

All incoming data to the olsrd daemon is handled by the *socket parser*. This entity can listen to multiple network sockets, which can be added in runtime. One socket is maintained per network interface running olsrd. For each of these sockets, a special *parser function* is registered. The parser function is called whenever data is available on the particular socket. Special parser functions for specific message types can be registered and added dynamically. If no function is registered to handle a message type, the packet parser forwards the message according to the *default forwarding algorithm* in OLSR.

As OLSR is a table driven routing protocol, updated information is kept in tables, or *information repositories*. All information about the current state of the network and quality of links are described in these tables. The different parser functions both update the information in the tables and makes use of stored information to process messages. To avoid duplicated packets, the forwarding function relies on the *duplicate table*, which stores all recently processed packets.

The event scheduler in olsrd runs different registered tasks at given intervals. If a certain packet should be transmitted within a defined time interval, a dedicated function can be registered with the scheduler.

7.2.2 Configuration

Olsrd is fairly easy both to configure and run. The daemon uses a human-legible configuration file, which is read by the process at startup. The configuration file defines the basic behavior of olsrd regarding which network interfaces should run OLSR, different message emission intervals to use, auxiliary plugins to load, and other parameters according to the RFC, such as link hysteresis and MPR willingness.

Altering the parameters in the configuration may not, however, cover all possible special modes of operation. The olsrd implementation is based on open source C code, which is relatively easy to understand, alter and extend for any experienced software developer. The most prominent feature of the olsrd implementation is, however, the *plugin interface*. The plugin interface enables extension of the protocol *without* altering the core code of olsrd. The plugin interface is a major part of the implementation, and perhaps one of the primary reasons for the popularity of the olsrd daemon.

7.3 Olsrd plugins

The olsrd implementation supports dynamically loaded libraries for auxiliary functions. These extensions are enabled using the generic plugin interface [90]. Via the plugin interface, a third-party programmer can create extensions to adjust, extend, or exploit different functionality in olsrd as shown in figure 7.2. Such extensions can for example utilize the scheduler inside the daemon to invoke new functions on timed events, access different variables, or even altering the routing table. By using special parser functions, the plugin can alter both incoming and outgoing messages. For a software programmer, the plugin interface gives access to intercept or change current operation of OLSR using a plugin instead of altering the inner code structure.

Olsrd plugins can be categorized in two groups; (i) plugins that extend or change functionality in OLSR itself, and (ii) plugins that exploit the MPR flooding function in OLSR to disseminate its own message types.

The first category of plugins can be used to extend OLSR to provide QoS routing, to enable secure routing, or to extend the routing daemon to include link layer information. The second category of plugins can be used by an upper layer application in order to extend OLSR with auxiliary message types, which can be parsed by the plugin. Auxiliary messages can be used for a variety of purposes: Provide name service in an ad-hoc network, distribution of encryption keys or dissemination of *service discovery* information.

The plugin interface to olsrd gives a MANET developer some major advantages:

1. The interface provides backward compatibility—there is no need to change code in olsrd itself when adding auxiliary functions.
2. Since the original olsrd code is not touched, the plugin can be licensed under any term.
3. The plugin can theoretically be implemented in any programming language.
4. The default forwarding algorithm in OLSR will forward unknown packet types according to the MPR scheme.

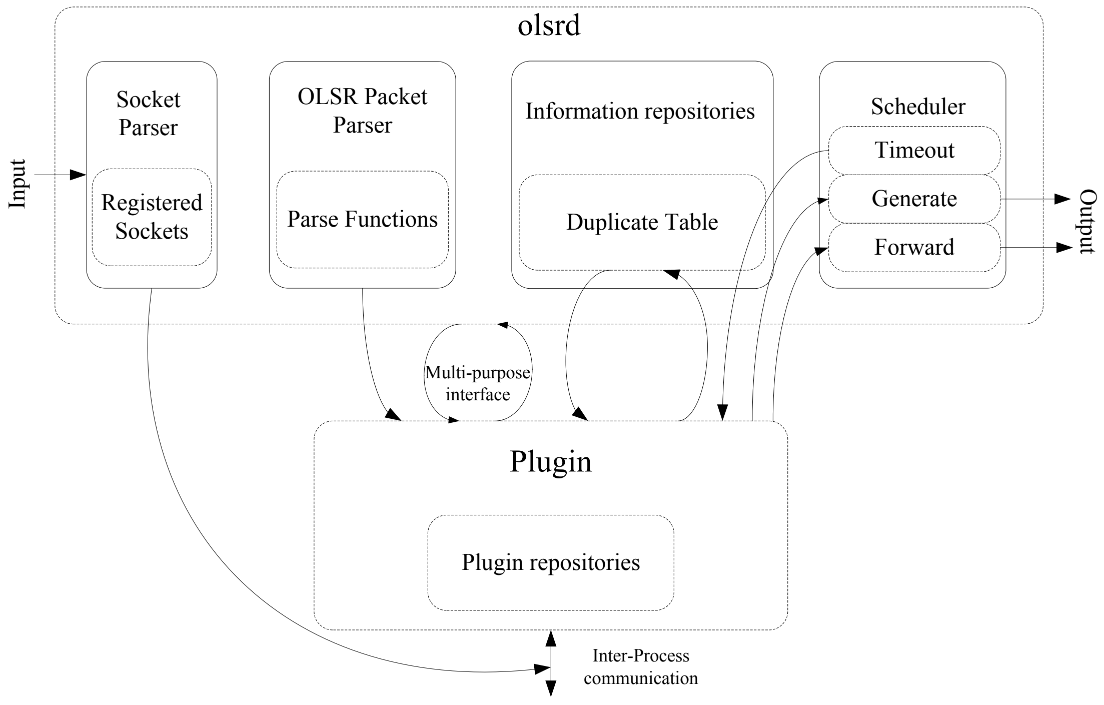


Figure 7.2: An olsrd plugin can intercept or change current operation of OLSR. Figure based on [89]

A great variety of different plugins exist. Some are created by the olsrd team and are included in the implementation found on the web site. Other plugins are part of different research projects and are described in papers, but are not part of the olsrd code. The list presents some of the available plugins:

- *Basic Multicast Forwarding Plugin (BMF)*. The Basic Multicast Forwarding Plugin floods IP-multicast and IP-broadcast traffic over an olsrd network. In order to optimize the flooding of multicast and local broadcast packets to all the hosts in the network, the Multi-Point Relays (MPRs) as identified by the OLSR protocol are used. A history of packets is maintained in order to prevent broadcast storms. Only packets that are classified as new to the process are forwarded. The plugin and its source can be downloaded as a part of the olsrd source [71].

- *HTTP Mini-server Plugin*. This plugin implements a small HTTP server that can be accessed from a browser. The plugin returns a HTML formatted page, which contains detailed process information from olsrd. The information provided includes detailed link status for all links and neighbors, all olsrd routes in the kernel, and local configuration. The plugin is included as a part of the olsrd source [71].

- *Power Status Plugin*. The Power Status Plugin gathers power information from the battery of a mobile node, and distributes the information to other olsrd-enabled nodes in the network. The plugin is described in [90], and although it is not compatible with the latest versions of olsrd, it still works as a good basis when designing new plugins.

- *Nameservice Plugin*. This plugin is a simple DNS replacement for OLSR networks and distributes host name information over OLSR. Every node that runs the plugin can announce different name-IP couplings via the plugin. These names can be its own host name, names of other IP addresses associated with HNA, and names resolved from an Internet DNS. The plugin is included as a part of the olsrd source [71].

- *Dynamic Internet Gateway Plugin*. This plugin checks dynamically whether the local node has an Internet connection or not. The plugin updates the local HNA information announced by the local node, facilitating Internet connectivity for other olsrd nodes in the network. The plugin is described in [90].

- *Encap Plugin*. This plugin includes a route management protocol for multi-homed wireless mobile nodes. The plugin facilitates low handover time when a mobile node switches between local access point or Internet Gateways by taking use of HNA information announced by the gateway nodes [22].

The above list effectively illustrates the variety of plugins that can be created to extend core OLSR functionality. The rest of this chapter will introduce a new member to this list—the Mercury Service Discovery Plugin.

7.4 Service Discovery as a plugin to olsrd

There are several advantages by implementing service discovery as a plugin to olsrd:

- *MPR Flooding*. Multicast in MANETs is still at the research stage (no standard is defined) and is thus an open issue. By using the previously defined cross-layer design and take advantage of message flooding using the Multi-Point Relays (MPRs), we have an efficient message dissemination scheme available without the use of IP multicast.

- *Piggybacking*. The service discovery message is defined as a separate message type, in addition to the built-in message types such as HELLO, MID, TC and HNA. A service discovery request may therefore be transmitted alone, or be piggybacked on one of the built-in message types. When piggybacking, bandwidth is saved since several messages are transmitted encapsulated in one single IP/UDP header.

- *Transparency*. Using the unified OLSR packet format, the OLSR standard provides extensibility of the protocol without breaking backwards compatibility. This feature gives a unique possibility to disseminate service discovery information transparent through intermediate nodes even if the nodes do not support the service discovery extension.

- *Availability to OLSR repositories*. As outlined in Figure 7.2, an olsrd-plugin has access to all variables and repositories inside olsrd. We can take advantage of this feature for several purposes: Our path-aware algorithm can exploit the local routing table when services are requested locally to avoid false positive lookups in the cache. Furthermore, the plugin can utilize the table of symmetric neighbors to make sure that no services are added to the local cache unless the node providing the service has a link considered stable by OLSR. In addition, other useful functions such as memory cleanup and socket handling in olsrd are available. The use of these existing and well proven functions avoids duplication of similar tasks, it reduces the complexity of the plugin implementation, and enhances both readability and stability.

7.4.1 Implementation overview

Even if the olsrd daemon itself is programmed in C, an auxiliary plugin can be written in any language that can be compiled to a dynamic loadable library. The Mercury Service Discovery Plugin is implemented—as the daemon—in C. There are two reasons for the choice of C as the programming language: (i) In order to make the interface to olsrd clean and easy to understand, C was chosen to avoid conversions and type casts between two different languages, and (ii) C has very few dependencies, and should therefore be easy to port to other platforms and operating systems in the future.

Even if the entire source code for the service discovery plugin is rather compact, and consists of less than 2000 lines, it is not included in this document but is available at [27]. Essential details in the code are explained in the subsequent sections. The implementation consist of a `/src` folder containing the following files:

```
mercury_plugin.h
mercury_plugin.c
bloom.c
bloom.h
Makefile
```

The first two files in the listing contain all service discovery functionality. The next two files contain the bloom filter algorithms, and can easily be replaced with other data structures if desired. The source should be placed in the `/lib` folder of the olsrd source. Building and installation is done by running the Makefile:

```
:~/src$ make
:~/src$ make install
```

The commands above compiles the Service Discovery code as a shared object:
`olsrd_mercury.so.x.y`, and places it under `/usr/lib`. The plugin is loaded by defining the library in the configuration file `/etc/olsrd.conf` according to [89].

7.4.2 Plugin architecture

The Mercury Service Discovery Plugin and the peripheral connections are shown in Figure 7.3. The main building blocks of the plugin are:

- *Repositories*: The repositories are tables that store both own services that are advertised, requested services, and foreign services advertised by other nodes.

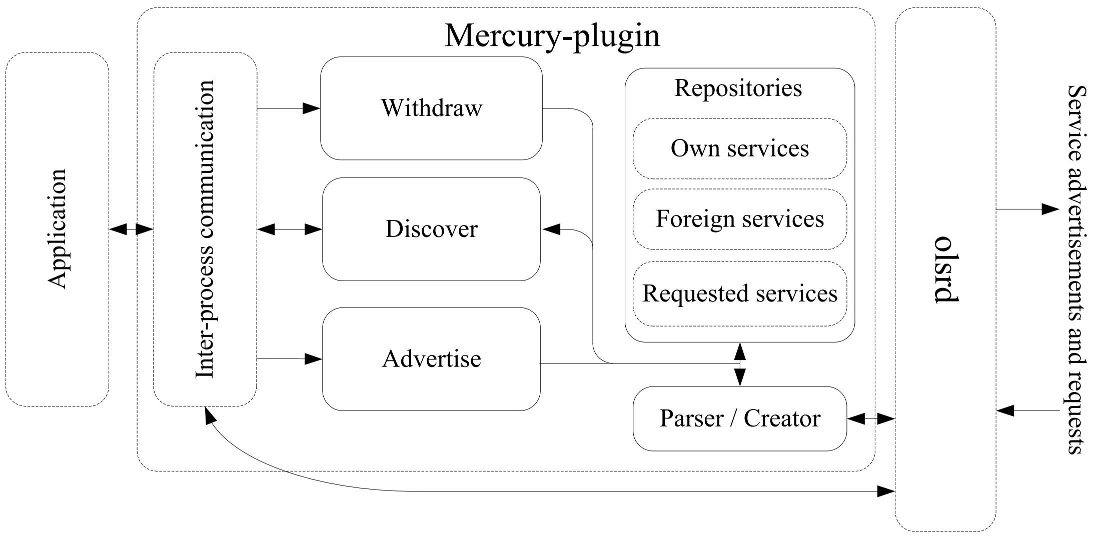


Figure 7.3: The main building blocks of the Mercury Service Discovery Plugin

- *Packet parser and creator*: The message parser function intercepts incoming Service Discovery Messages, while a creator function creates and prepares service discovery messages for transmission.

- *Inter-Process Communication*: Inter-process Communication (IPC) is a way to provide two-way communication to an upper-layer application.

- *Interface to olsrd*: Different interfaces to olsrd provide functionality to load and shut down the plugin, transmit packages, and deal with sockets.

- *Service access functions*: The service access functions deal with service advertisement and request. Those functions also perform sub-tasks using either IPC to applications, interfaces in olsrd or by accessing the local repositories.

7.4.3 Repositories

The repositories are lists that store certain information. Inside the service discovery plugin, there are three such repositories:

Own services In this list, all the different services offered by the local node are stored. The service descriptors are stored as plain text. This makes it possible to search or withdraw services by their service name. One single entry in the list exists per service offered. Upon sending a service advertisement, all the service descriptors in the list are encoded in one single Bloom filter. The services in this list persist until an upper-layer application *withdraws* the service. When the olsrd daemon is restarted, the list is cleared.

Foreign services In this list, all the services offered by other nodes are stored. Each entry consists of the Bloom filter advertised by a foreign node and its current IP address.

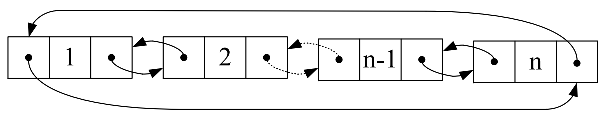


Figure 7.4: A two-way circular linked list. Every entry in the list holds two pointers, and the last entry in the list points to the base element.

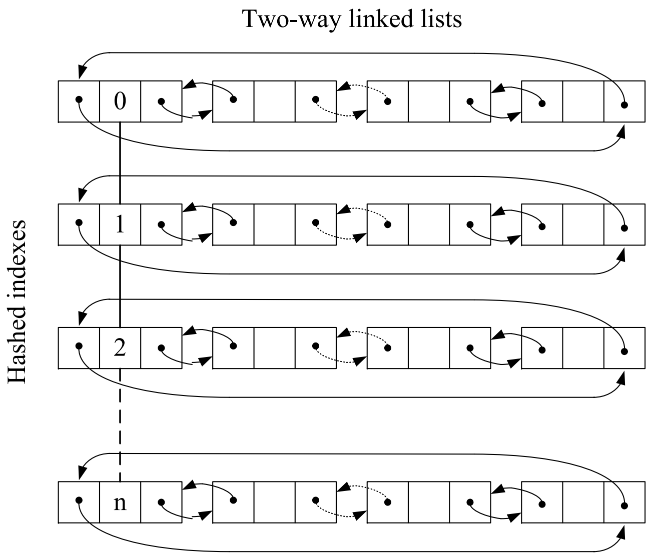


Figure 7.5: An initial hash value indexes one of n two-way circular linked lists.

Requested services This list stores all the services this node is requesting. If a successful service reply is not received within a predefined time, the request eventually times out and the list entry will be deleted.

All repositories are implemented as two-way circular linked lists. Every entry in these lists holds two pointers: a pointer to the *previous* data element (in this case service descriptor or node), and a pointer to the *next* element. The last entry in the list points to the base element, which makes the list circular (Figure 7.4).

The benefit by this data structure is that the order of the linked elements can be different from the order that the data elements are stored in memory. This allows the lists to be traversed in any order, and permits insertion and removal of entries at any point in the list. Another advantage of a linked list in contrast to a conventional array is that entries can be inserted indefinitely. An array will eventually either fill up or need to be resized.

The repository containing the list of foreign nodes could be as large as the total number of nodes in the network. As an effect, a standard linked list may be cumbersome to search due to its length. To solve this issue, the list of foreign services is implemented as an array of several two-way circular linked lists. The root element of each linked list is an element of an array where the index is a hash value of the IP-address of the service provider (Figure 7.5).

7.4.4 Packet parser

The parser system consists of three components:

- A definition of the Mercury service discovery message.
- A function to create messages.
- A function to parse incoming messages.

The definition of the Mercury service discovery message is illustrated in Figure 4.4 on page 29 and serves as a base for the implementation. In the plugin, this message is defined as a simple C-struct:

```
struct mercurymsg
{
  olsr_u8_t     type;
  olsr_u8_t     length;
  olsr_u16_t    spare;
  unsigned char filter[FILTER_LENGTH];
};
```

The special olsr datatypes such as `olsr_u8_t` and `olsr_u16_ct` are defined in `olsr_types.h` The type of the message can be either `MSD_ADVERTISEMENT`, which indicates a message containing one or more *service advertisements*, or `MSD_REQUEST`, which means that the message contains one or more *service requests*.

In order to inform olsrd that a new message is defined, a message parsing function is registered with olsrd when the plugin is initialized:

```
olsr_parser_add_function(&olsr_parser, MERCURY_PACKET, 1);
```

When a new service discovery message is received by olsrd, this is identified uniquely by the Message Type field set to `MERCURY_PACKET`. Olsrd then calls the plugin function `olsr_parser` By doing this, olsrd hands over the responsibility to parse the message to the plugin.

```
void
olsr_parser(union olsr_message *m,
    struct interface *in_if,
    union olsr_ip_addr *ipaddr)
```

Prior to message parsing, the plugin verifies that the originator of the message is considered a symmetric neighbor by calling a checkup function in olsrd. The plugin also verifies that this packet is not previously processed by checking the duplicate table in olsrd:

```
if(check_neighbor_link(ipaddr) != SYM_LINK) {
    return;
}
if(olsr_check_dup_table_proc(&originator, seqno){
    process_message(m);
}
```

When all parsing and handling of the service discovery message is performed, the message is forwarded using MPR flooding:

```
olsr_forward_message(m, &originator, seqno, in_if, ipaddr);
```

If the message is considered valid, the message is processed by one of the *service access functions* described subsequently.

7.4.5 Service access functions

The service access functions handles requests and advertisements both from the external network via olsrd, and from applications running locally on the node—connected via Inter-process communication (described in section 7.4.6). The access functions controls the internal repositories by updating and deleting entries when needed. As service descriptors may be handled either as clear text or as a Bloom filter, the following vocabulary is established as a reference: A service request is named S_R, a service advertisement is named S_A. One or more service requests and advertisements encoded using Bloom filters are named $B(S_R)$ and $B(S_A)$ respectively.

Incoming service messages are handled by one of two functions based on the type-field of the message:

- *Advertisements:* An advertisement $B(S_A)$ received from a network node is immediately added to or updates an existing entry in the ”Foreign services” repository. If $B(S_A)$ matches one of the entries in the ”Requested services” repository, a message is sent to all IPC connected applications.
- *Requests:* A request $B(S_R)$ received from an external node immediately query the ”Own services” repository. If a matching service S is found, the node replies with a service advertisement. A service advertisement $B(S_A)$ is created as a Bloom filter containing all the services in the ”Own services” repository and sent.

Both incoming requests and advertisements are, regardless of their content, forwarded using the MPR flooding technique. In order to avoid loops, the plugin verifies that this packet is not previously processed by checking the duplicate table in olsrd.

Requests from local applications are received via Inter-process communication. The plugin supports three different requests:

- *Service Request:* A service request S_R is immediately hashed as a Bloom filter to $B(S_R)$. Then the filter is queried in the ”Foreign services” repository cache. If no match in this repository is found, the service request S_R is matched against the ”Requested services” repository to check wether a request is recently performed regarding the same service descriptor. If this is not the case, the service descriptor S is added to the ”Requested services” repository. Then, a new Service Request message $B(S_R)$ is created as a Bloom filter containing *all* the services in the ”Requested services” repository, and sent.
- *Service Advertisement:* When an application is advertising a service, the service descriptor S is added to the ”Own services” repository. Then a service advertisement $B(S_A)$ is

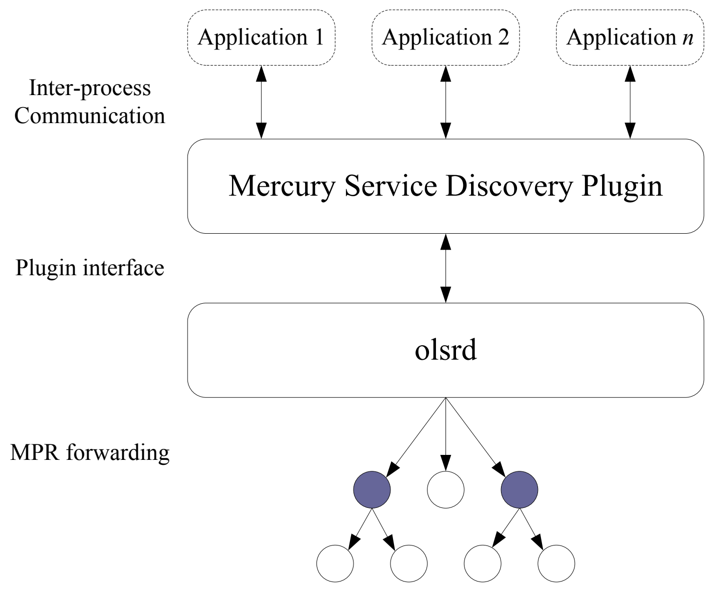


Figure 7.6: A number of applications, n, are connected to the plugin. Requests and advertisements are disseminated using MPR forwarding in OLSR.

created as a Bloom filter containing *all* the services in the ”Own services” repository, and sent.

- *Service Withdrawal:* When an application shuts down or for some reason can not provide a service S anymore, it shall withdraw the service. The service descriptor S is then removed from the ”Own services” repository. Subsequently, a new service advertisement $B(S_A)$ is created as a Bloom filter containing all the *remaining* services in the ”Own services” repository, and sent using the MPR flooding technique. Notice that the advertisement is sent even if the ”Own services” repository is empty and the resulting Bloom filter is NULL. To speed up the service discovery process for nodes dependent of the recently withdrawn service S, a service request message $B(S_R)$ containing a Bloom filter of the withdrawn service is created and immediately sent, piggybacked to $B(S_A)$. As a result of this technique, any node providing a service S will respond to the request $B(S_R)$ and nodes dependent of S will contact the new service provider immediately.

7.4.6 Inter-process communication

To allow communication between the plugin and user applications, an Inter-process communication function (IPC) is created. The IPC communication is enabled using TCP/IP via the loopback interface. The plugin allows several simultaneous applications to connect via IPC as illustrated in Figure 7.6. The number of simultaneous applications that can connect to the plugin is limited by `MAX_IPC_CLIENTS` defined in `mercury_plugin.c`.

A connection to the plugin is established simply by creating a TCP socket to localhost (usually `127.0.0.1`) on the port number `IPC_PORT` defined in `mercury_plugin.c`. The Interprocess Communication interface can be tested using a telnet client, provided by most operating systems. Given that the `IPC_PORT` is `8888`, the connection is done by typing:

```
telnet localhost 8888
```

When an IPC-socket connection is established, the IPC interface provides a few simple text based commands, which the application can put to use. The default setup of a complete command is:

```
<Command> <ServiceName> <Attribute>
```

The different commands facilitate *advertisement*, *withdrawal* and *request* of services:

Advertisement
A service advertisement is performed by giving the command: `ADVR <Service>`. An example of usage: If a chat-application with the name ”Chatclient” starts, it advertises itself to the network. Assuming that the IPC-socket is established, this is done simply by giving the command:

```
ADVR Chatclient
```

Withdrawal
When an application either shuts down or for some other reason cannot provide the service anymore, it withdraws the service. This is done simply by giving the command: `WTDR <Service>`. In our chat client example, the command is:

```
WTDR Chatclient
```

Request
A service request is performed by giving the command: `RQST <Service> [ANY,ALL]`. If an application want to retrieve the IP-addresses of *one* of the printers in the network, it sends the following command: `RQST Printer ANY`. On the other hand, if the application wants to retrieve *all* of the chat client in the network, it uses the following command:

```
RQST Chatclient ALL.
```

Output
The plugin also has the ability to provide data to the application. The plugin will respond with `OK` if one of the commands is understood and action is performed. If a requested service is found in the network, the plugin will respond with:

```
SERVICE FOUND: <Service> AT <IP> <Time>.
```

Say that the application has asked for *one* of the printers in the network, like the above example, the reply may look like this:

```
SERVICE FOUND: Printer AT 192.168.0.4 (0.131s)
```

7.4.7 Summary

This chapter has described the implementation of the Mercury Service Discovery protocol aimed for real-life usage. The implementation is made using the plugin interface of olsrd and is available at [27]. The Mercury plugin can be used in a wide range of applications. Appendix C shows how SIP user agents can be discovered in an ad-hoc network using the Mercury plugin.

Chapter 8

Simulation methods

Two paradoxes are better than one; they may even suggest a solution.

Edward Teller

This chapter provides background information and addresses some pitfalls and frequent source of errors when simulating mobile ad-hoc networks. Finally, the chapter explains my choice of simulation and validation models.

8.1 Performing valid measurements

The key questions to answer when evaluating a service discovery protocol (or any ad-hoc network protocol) are:

- What to measure?
- How to measure?
- How to evaluate the measurements?

The questions above must be addressed prior to any simulation study. Also, in order to measure the behavior of any protocol by simulation, valid scenarios must be established. Scenarios must be *realistic* in order to cover the future use of the protocol, but must be kept *simple* to be enable to isolate and test one particular feature at a time.

The features of Mercury that is tested in this thesis are:

Bloom filter The implementation of the Bloom filter is tested in order to verify that the implementation behaves according to the mathematic theory.

Caching The effect of caching is examined to measure performance gains and to detect any side effects.

Delay The delay (the time consumed) to perform service discovery in different networks is evaluated.

Overhead The overhead (the number of bytes) induced by the service discovery process is examined using different network topologies.

In order to measure the above parameters one can take advantage of both static and mobile scenarios. *Static scenarios* are easy to set up and to repeat and are feasible when distinct features of the protocol are measured. Scenarios including node *mobility* are more realistic. As the employed mobility model greatly affects the performance of the simulated protocol, a *realistic* model must be used.

8.2 Obtain a realistic dynamic topology

In order to obtain confident results when simulating new protocols and algorithms for ad-hoc networks, it is imperative to use a mobility model that is suitable for the target application. Huang et al. have addressed this point by creating mobility particular to simulate first responders at an incident scene [40]. Additionally, T. Camp et al. have proved that the performance of an ad-hoc network protocol can vary significantly with different mobility models [11].

Researchers agree that only by using an appropriate mobility model that closely matches the real world scenario, one can evaluate and determine the effects of a given protocol. Two main approaches exist when choosing mobility models for mobile network simulation:

- To use traces or tracks from *real-world patterns*.
- To take advantage of *synthetic models*.

Traces can be collected by equipping people and vehicles with GPS-loggers when performing a realistic operation. Even if traces provide more accurate information for a given scenario than synthetic models, they are seldom used in ad-hoc network research. In [42], real traces are collected and used to simulate a vehicular ad-hoc network. M. Kim et al. provide a way to collect traces for simulations by gathering logs from Wi-Fi access points [49].

There are obvious reasons for the lack of published results using real-world tracks: First, traces are hard and expensive to obtain—especially for a large number of nodes. Second, it may be difficult to foresee a specific scenario, and thus impossible to collect valid traces. In such cases, *synthetic models* are crucial. Several synthetic models exist to simulate ad-hoc networks such as random walk, random waypoint, random direction and probabilistic random walk [11].

In order to evaluate the Mercury service discovery protocol, I have used *both* a synthetic model and real traces. This chapter compares two different routing protocols using Random Waypoint Mobility Model as an example of the most popular synthetic model, plus tracks collected from a real-world exercise. The reason for doing this initial comparison is twofold:

- Examine how the chosen mobility model influences the performance of the routing protocol.

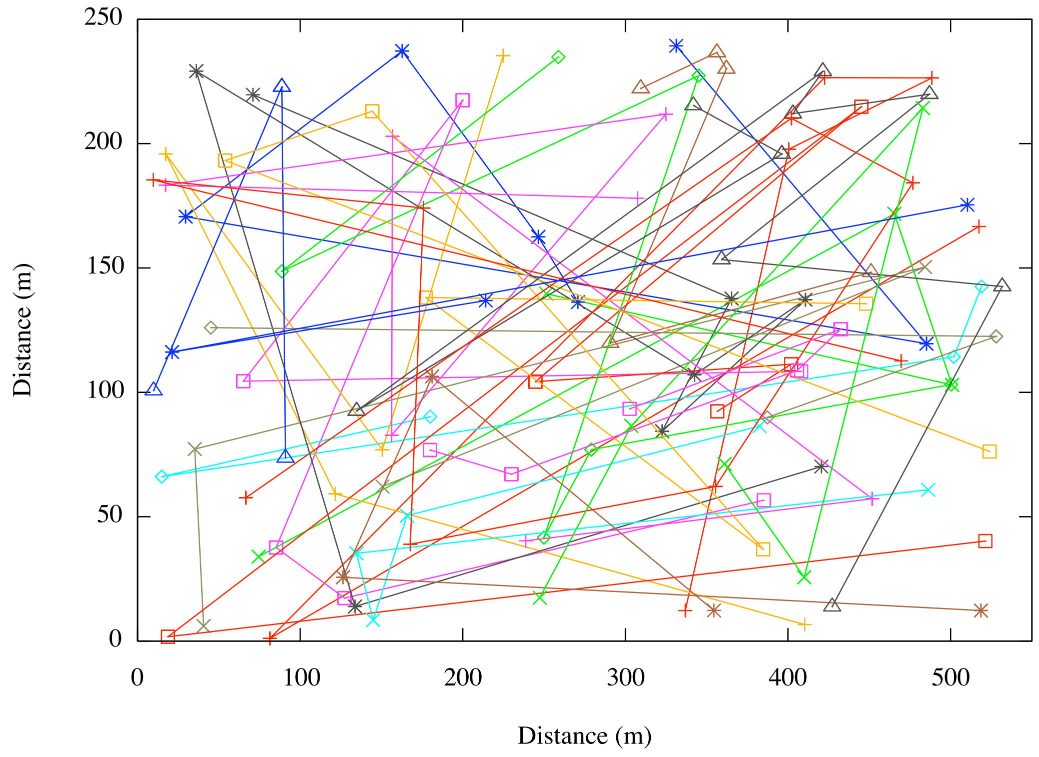


Figure 8.1: Traveling pattern of 22 mobile nodes following Random Waypoint Mobility Model.

- Obtain an understanding of the effects of choosing one routing protocol in favor of another. Mercury is a cross layer service discovery protocol, and its performance is therefore bound to follow the performance of the routing protocol.

The results in this chapter are used as a base to create valid scenarios for following tests of the Mercury service discovery protocol. The results also provide important knowledge for validating subsequent simulations.

8.2.1 Comparing synthetic mobility models and real-world traces

I created a simple scenario to compare the synthetic mobility model and real-world traces. The traces were collected from a tactical exercise with real soldiers. The exercise included 22 soldiers divided in three teams that first moved independently, and then they collaborated. The exercise area was 530 x 240 m, and included both forestry and a village area. All subsequent simulations are performed using the same area size and the same number of nodes.

Notice that two distinct features separate real-world tracks from synthetic tracks: *Obstacles* and *collaboration*. In the real life, users have to deal with obstacles such as buildings, constructions and vegetation. Users therefore move along paths and roads. Additionally, real users tend to cooperate and move in groups. Nodes simulated by the Random waypoint model do, however, take neither obstacles nor collaboration nto consideration.

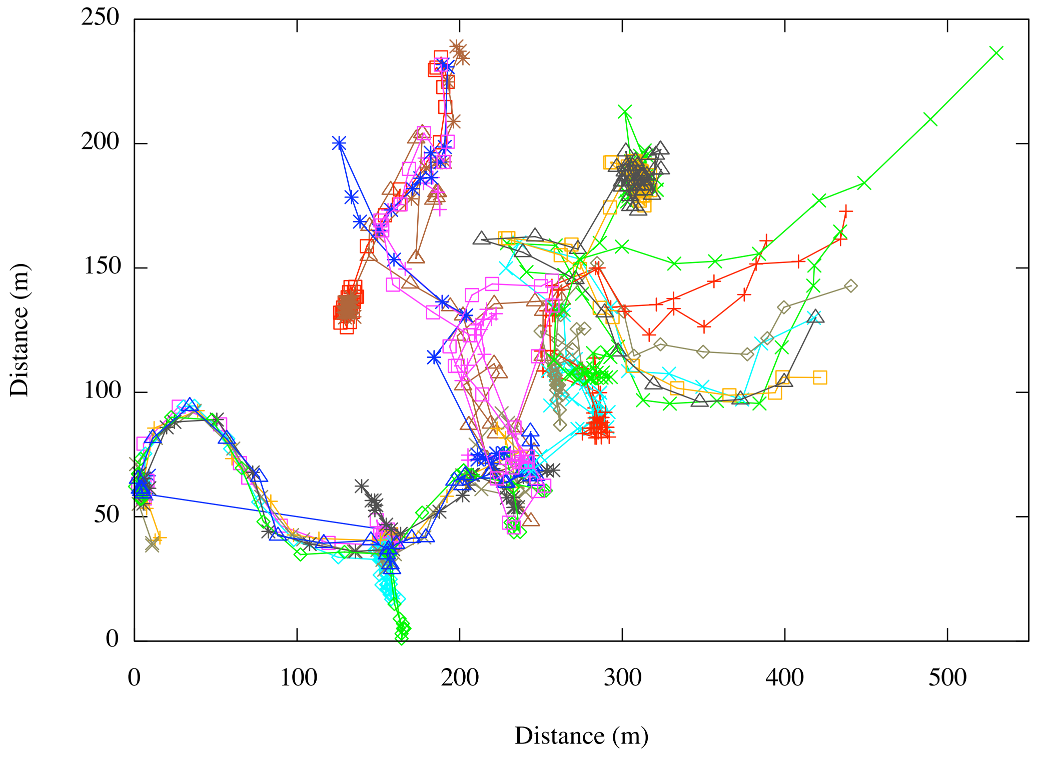


Figure 8.2: Traveling pattern of 22 mobile nodes following real position tracks.

The effect of this simplification is effectively demonstrated by figure 8.1 and figure 8.2, which show the movement patterns of Random Waypoint and the real traces respectively. We clearly see that the nodes are more evenly distributed across the area when the synthetic model is used compared to using real tracks.

It is expected that the two different mobility patterns will influence the performance of the chosen routing protocol. The following simulations will test this hypothesis.

8.2.2 Scenario description

The ns-2 network simulator [92] was used to perform the simulations. Two different routing protocols were included: the built-in AODV implementation and UM-OLSR from the university of Murcia [93]. Both AODV and OLSR used the default parameter settings as described in their corresponding RFCs 3561 [75] and 3626 [20] respectively.

The traffic pattern in the network was constant bit rate (CBR) connections, with fixed packet sizes of 50 bytes. Each of the 22 nodes transmitted one packet to each of the other nodes every 10s. The CBR connections were initialized after a warm-up time of 60s. The purpose of the 50-byte package was to simulate typical location service (GPS) messages, which is an important feature in tactical networks. For the sake of simplicity, no multicast feature was enabled.

50 different movement patterns were generated for the Random Waypoint model. For each run, the number of successfully received packets and the number of hops were logged in order to

Parameter	**Value**
Simulator	NS-2.31
OS	MAC OS 10.5.2
Simulation Time	1550s
Simulation Area	530 x 238 m
# Nodes	22
Transmission Range	100m
MAC	802.11
Movement Model	Random Waypoint / (real-world)
# Patterns	50 (1)
Node speed	$1.0\frac{m}{s}$ / (real)
Pause Time	0.0s / (real)
Routing Protocol	UM-OLSR / NS-2.31 AODV
CBR Sources	22
Data Payload	50 bytes
Packet Rate	0.1 packets / sec
Traffic Pattern	peer-to-peer

Table 8.1: Setup of the simulation. Numbers for the real-world trace simulation in parenthesis.

			Number of hops (% of traffic)									Avg
Protocol	**Mobility**	**Lost packets**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
AODV	Real	4.5%	51.9	27.0	11.6	3.9	0.9	0.1	0.0	0.0	0.0	1.6
AODV	RWPT	29.2%	25.5	18.4	12.4	7.4	4.1	2.0	0.8	0.3	0.1	1.7
OLSR	Real	5.6%	60.3	25.9	7.1	1.1	0.1	0.0	0.0	0.0	0.0	1.4
OLSR	RWPT	25.0%	30.1	23.2	12.7	5.7	2.3	0.7	0.2	0.0	0.0	1.6

Table 8.2: Amount of lost packets, and distribution of traffic regarding to the number of hops. Both real tracks (Real) and random waypoint (RWPT) are used.

compare traffic distribution, node distribution, packet loss and delay. The results were averaged and the 95% confidence interval was estimated and given in the figures. Due to time and resources available, real world tracks from only one (albeit fully realistic) exercise was used. Simulation setup is given in Table 8.1.

8.2.3 Traffic distribution

I wanted to examine how the CBR traffic was routed in the network using different routing protocols and mobility models. Figure 8.3 and Table 8.2 illustrate how the different mobility models influence the number of hops necessary to transmit CBR packets end-to-end in the scenario. Using real tracks, the number of hops necessary to establish connection between any pair of nodes never exceeds six, while the Random Waypoint model yields longer paths.

Table 8.2 also reveal that—regardless of the routing protocol—a great number of CBR packets are lost in the Random Waypoint scenario. Comparing routing protocols and mobility models, we see that the mobility model has a greater affect on the performance than the choice of routing protocol.

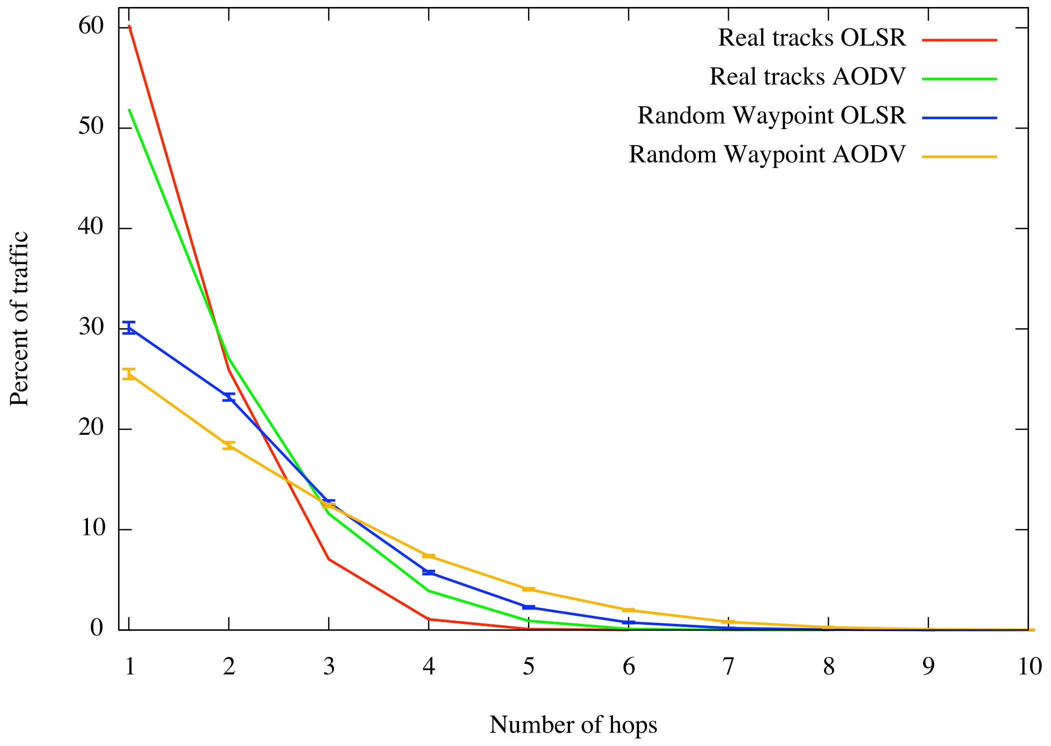


Figure 8.3: Amount of traffic transmitted with respect to number of hops.

8.2.4 Node distribution

In real-world networks, users are expected to form groups and move united. One way to measure this effect is to count the number of neighbor nodes (1-hop) and the number of routed nodes. A high percentage of neighbor nodes implies that the nodes are formed in groups. A high number of routed nodes implies (obviously that the routing works and) that the groups are not clustered beyond radio transmission reach.

Figure 8.4 shows the average percentage of neighbor nodes (1-hop) when changing routing protocol and mobility model. As expected, the number of neighbor nodes is independent of the choice of routing protocol. Comparing the mobility models isolated, we observe that the real-world track model leaves more nodes in the one-hop proximity. This is expected, as real-world nodes collaborate and move together. This effect is not considered in the Random Waypoint model.

By examining all CBR packets transmitted from each node, the number of accessible routed nodes and the average hop-count can be found (Figure 8.5 on page 70). The figure reveals two important findings: (i) when the real nodes are clustered in groups (as seen in the first 400s), coverage is reduced compared to Random Waypoint. (ii) When real nodes collaborate (as with the last 1100s) coverage increases compared to Random Waypoint. Hence, Random Waypoint underestimates both group clustering and node collaboration.

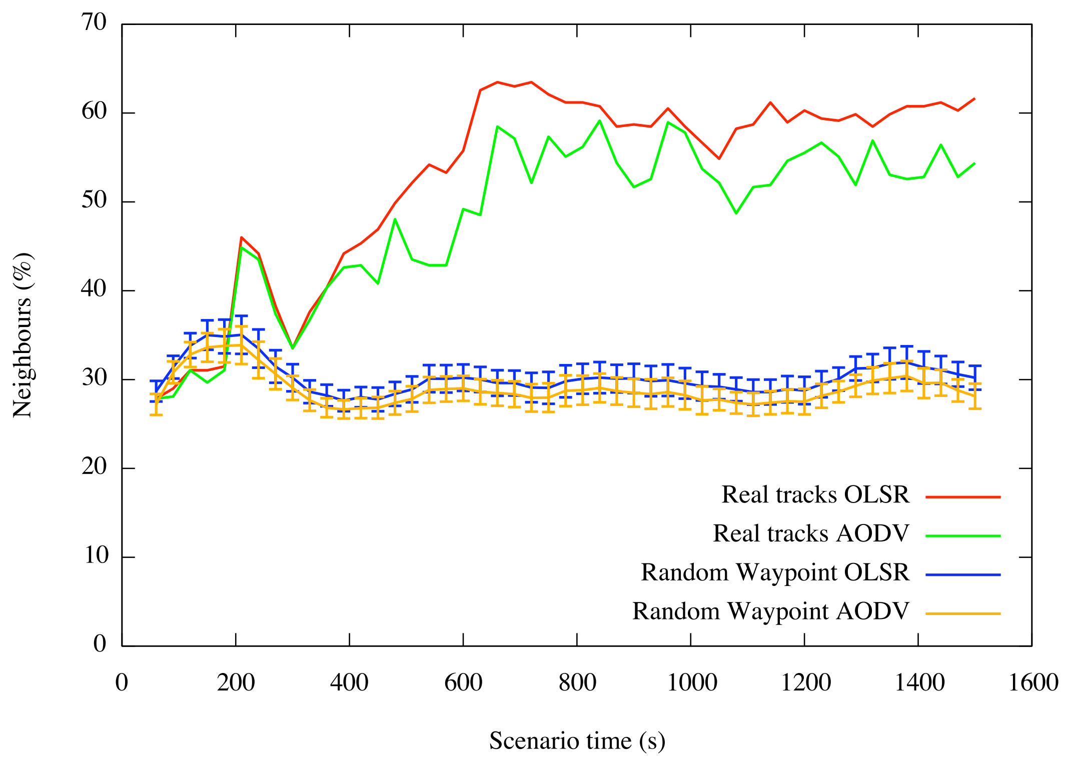


Figure 8.4: Average number of neighbors during the 1550s run.

8.2.5 Delay

The time-delay between transmission and receiving a packet is another interesting feature. Figure 8.6 reveals that the delay for packets traversing between one-hop neighbors is mainly dependent of the routing protocol and not the mobility model. As AODV is a reactive protocol, it is expected to yield longer delays than the proactive OLSR counterpart.

Figure 8.7 illustrates the end-to-end delay for all packet transmissions. We clearly see that, since the average hop count is lower in the real-track model, the delay is reduced compared to the Random Waypoint model. Thus, as Random Waypoint treats each node independent, the model overestimates delay.

8.2.6 Conclusions

From the simulations conducted, the following conclusions are established:

- It is crucial to create a realistic scenario when performing performance evaluation. Real traces from a real exercise or test are preferred.
- OLSR performs better than AODV regarding end-to-end delay in the network for both Random Waypoint and when using real tracks. It is not the scope of this thesis to perform a thorough performance comparison of OLSR and AODV. The results, however,

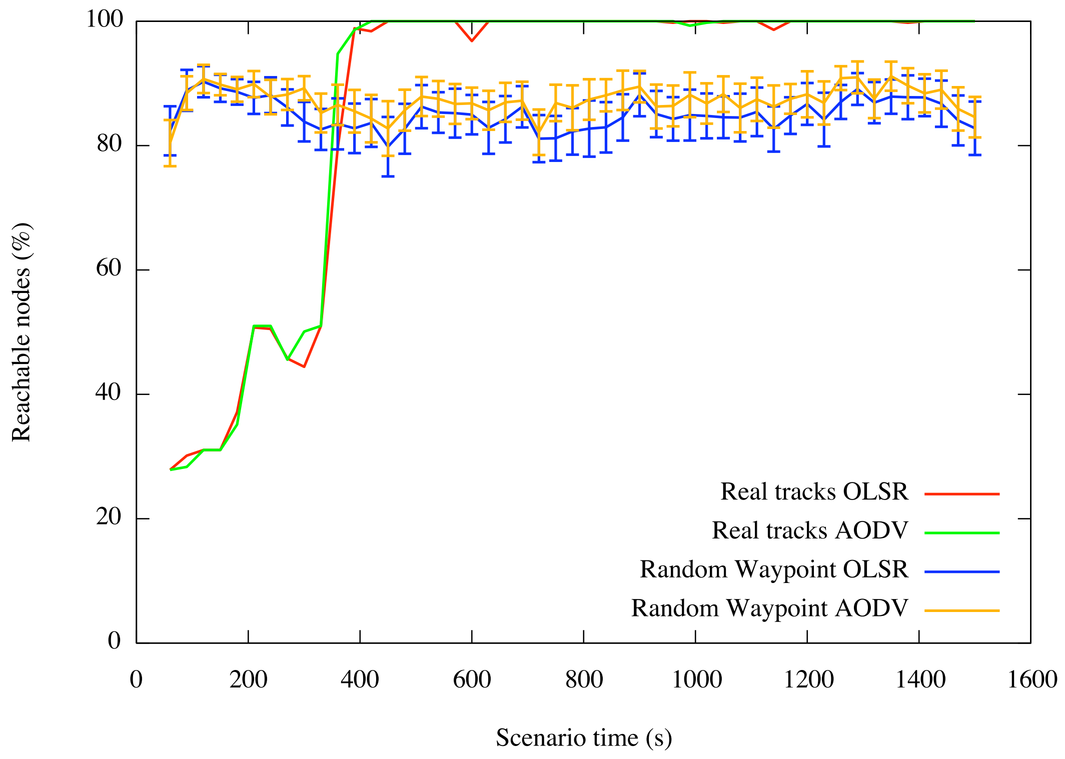


Figure 8.5: Average number of all accessible nodes during the 1550s run.

correspond well with previous research [21, 42].

- Random Waypoint treats each node independent and underestimates the relative dependence of the nodes. Hence, the routing protocol and any other protocol dependent on the routing protocol (such as cross layer service discovery) will perform different in the simulation compared to the real life environment.
- Random Waypoint overestimates delay compared to the real world tracks.
- Random Waypoint underestimates the variation in the topology.

As a summary of the above, I state that the mobility pattern will have a greater effect on the results from an ad-hoc network simulation than the choice of routing protocol.

8.3 Scenarios used in this thesis

Random Waypoint is the most common mobility model to validate ad-hoc network protocols—despite of the different weaknesses discovered [96] and the issues explained in this chapter. In this thesis, *different scenarios and mobility models* are used to evaluate the Mercury service discovery protocol to obtain confidence. All simulations and their results are given in the next chapter.

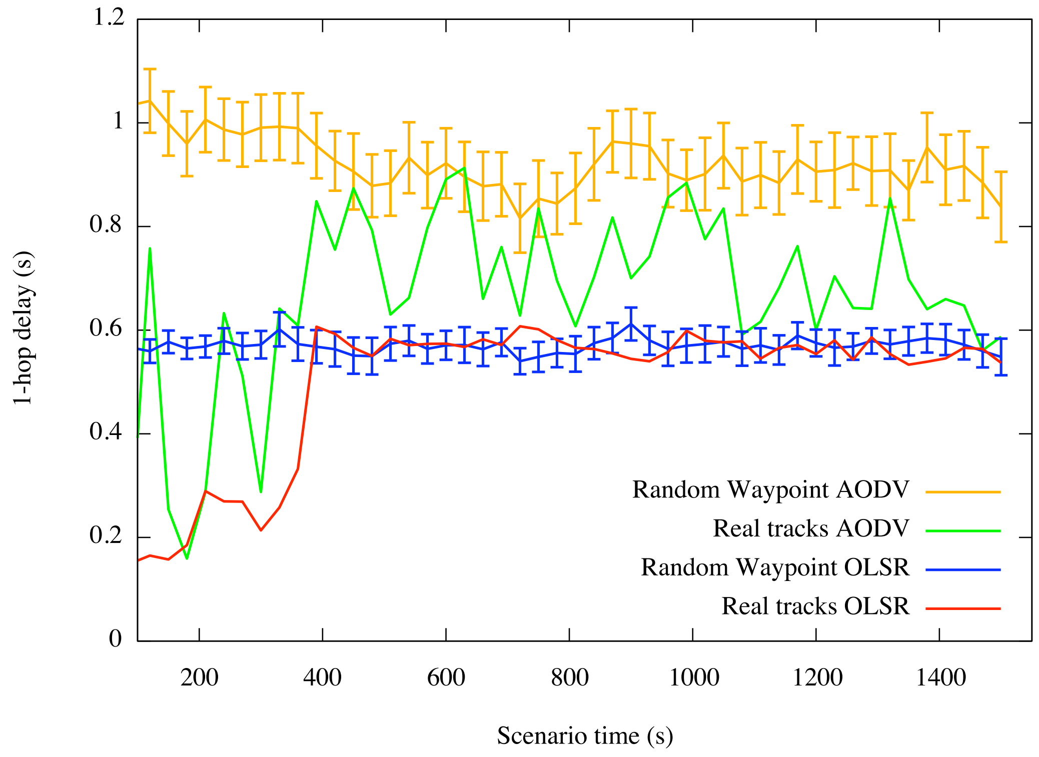


Figure 8.6: Average delay for one hop using different routing protocols and mobility models.

8.3.1 Static scenarios

Static scenarios with no mobility are used to make initial performance evaluation of the protocol. Distinct features such as the Bloom filter and overhead measurements are more practical to evaluate by static models. Static models are also used to make comparative simulations with real-world measurements and to compare Mercury with existing service discovery protocols.

8.3.2 Dynamic scenarios

Dynamic scenarios (with synthetic mobility) are used to evaluate the caching features of Mercury. A mobile scenario using real tracks is used to make a final and realistic evaluation of the protocol.

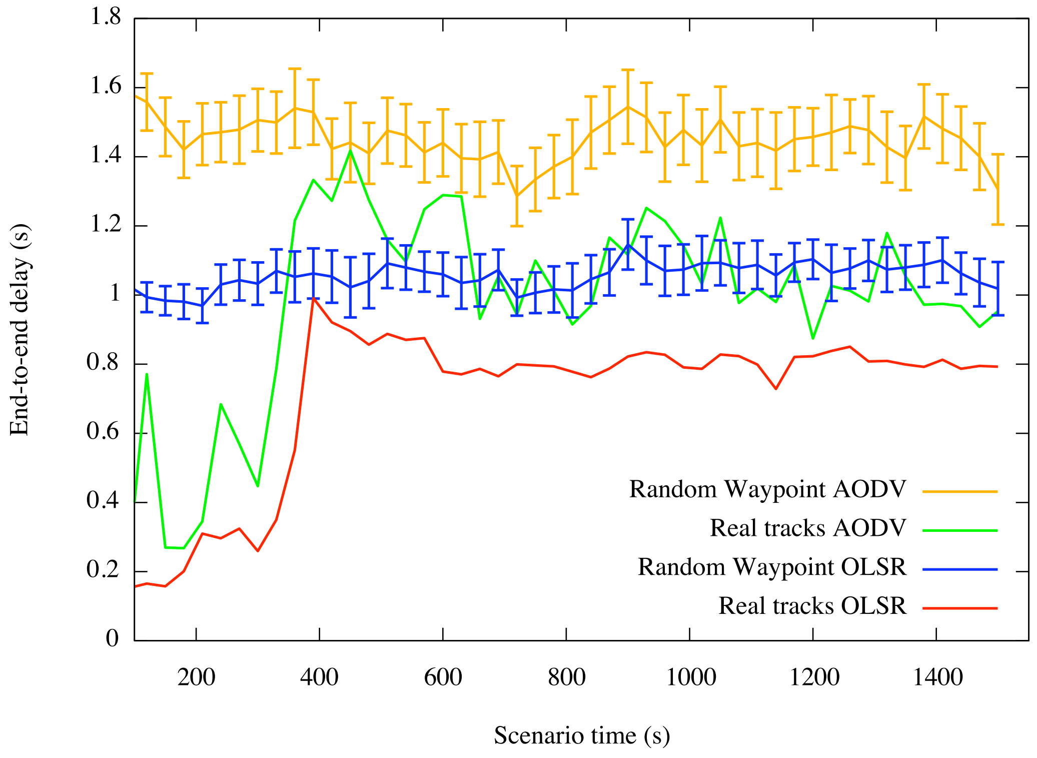


Figure 8.7: Average end-to-end delay using different routing protocols and mobility models.

Chapter 9

Simulations

What happens if a big asteroid hits Earth? Judging from realistic simulations involving a sledgehammer and a common laboratory frog, we can assume it will be pretty bad.

Dave Barry

In this chapter, simulations are performed to test the most prominent features of the Mercury service discovery protocol. Different static, mobile, and real-life scenario setups—as introduced in the previous chapter are used to test different features. The simulation setup, results and the conclusions are listed for each test.

9.1 Introduction

The ns-2 network simulator is used for all simulations. The code extension described in chapter 6 is included and compiled with the ns-2 code. The parameters listed in Table 9.1 are used for all simulations unless otherwise mentioned.

Parameter	**Value**
Simulator	NS-2.31
OS	MAC OS 10.5.2
Transmission Range	100m
MAC	802.11
Reflection model	Two Ray Ground
Movement Model	Random Waypoint
Routing Protocol	UM-OLSR
OLSR Settings	Default [20]

Table 9.1: Default setup for all simulations.

	Number of available services								
k	1	2	4	8	16	32	64	128	256
1	.0072	.0145	.0296	.0625	.1208	.2345	.4342	.7003	.8823
2	.0003	.0016	.0034	.0121	.0408	.1741	.3941	.7664	.9494
3	0	.0003	.0009	.0041	.0242	.1717	.4750	.9304	1.0000
4	0	0	.0002	.0016	.0173	.1632	.5258	1.0000	1.0000
5	0	0	.0001	.0010	.0100	.1585	.6101	.9619	1.0000
6	0	0	.0001	.0011	.0171	.2359	.8244	1.0000	1.0000
7	0	0	0	.0009	.0129	.2178	.8044	1.0000	1.0000
8	0	0	0	.0008	.0091	.2380	.8294	1.0000	1.0000
9	0	0	0	.0003	.0143	.3035	.9322	1.0000	1.0000
10	0	0	0	.0003	.0193	.3447	.9250	1.0000	1.0000

Table 9.2: Measured false positive probability using a 128 bit Bloom filter.

9.2 False positive probability of the Bloom Filter

Bloom filters are a major component of the Mercury service discovery implementation. The false positive property of such Bloom filters is described and evaluated analytically in 4.2.1. These analytic results represent the theoretic optimum, given a perfect hash function. No Bloom filter implementation is, however, expected to achieve the optimum value. It is important to evaluate how the Mercury implementation corresponds with the analytic results. If a correlation is found, the equations in chapter 4 can be used (with certainty) to estimate the impact of the false positive probability when altering one of the Bloom filter parameters (number of hash functions, width of the filter or number of services).

9.2.1 Description

The false positive rate was estimated for different combinations of k (number of hash functions) and n (number of services offered). In order to isolate the false positive feature, a static scenario was chosen, consisting of two nodes: A and B. A was offering a set S consisting of $n = \{1, 2, 4, 8 \ldots 256\}$ services using $k = \{1..10\}$ hash functions. For each combination of service number and hash functions, node B requested a set of 10000 different services which were intentionally not part of the set S. The false positive rate was calculated as the amount of service requests out of the 10000 requests yielding a positive reply. The width of the Bloom filter was kept constant at 128 bits.

9.2.2 Results

Table 9.2 shows the false positive rate measured by the simulations, and Figure 9.1 compares the simulated result with the expected false positive rate calculated by equation 4.1 on page 25. On average, the false positive rate of Mercury is 0.5 percentage points above the theoretic optimum. Increasing the number of service requests can reduce the variance observed in the figure.

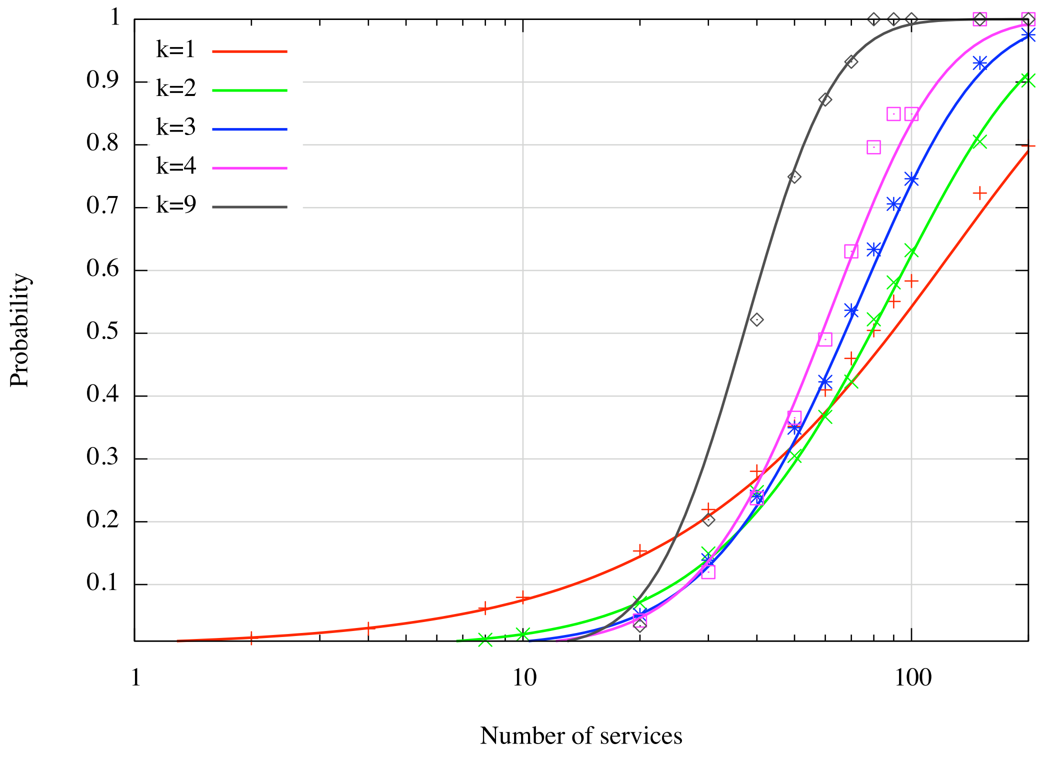


Figure 9.1: False positive probability of a 128-bit Bloom filter. Measurements from simulations (dots) compared with calculations (solid-drawn line) for each value of k.

9.2.3 Conclusions

The following conclusions are drawn from the results:

- The performance of the MD5 based Bloom filter in Mercury closely matches the theoretic optimum.
- Given the strong correlation between the theoretic performance and the measurements, future evaluations of the effect by changing Bloom filter values (hash functions, number of services, or filter width) can be performed with confidence using the equation 4.1.

9.3 Path-aware algorithm

The Mercury service discovery protocol utilizes caching of the advertised services in order to save overall network bandwidth. Local caching may however, lead to false positives if the advertised service exist in cache even if the node that advertised the service is not available anymore. Such false positive replies cause unacceptable delays and reduce user satisfaction. For this reason, Mercury includes a Path-aware scheme as described in 4.5.1 on page 31. Simulations were performed in order to reveal any possible benefits by implementing the proposed caching scheme.

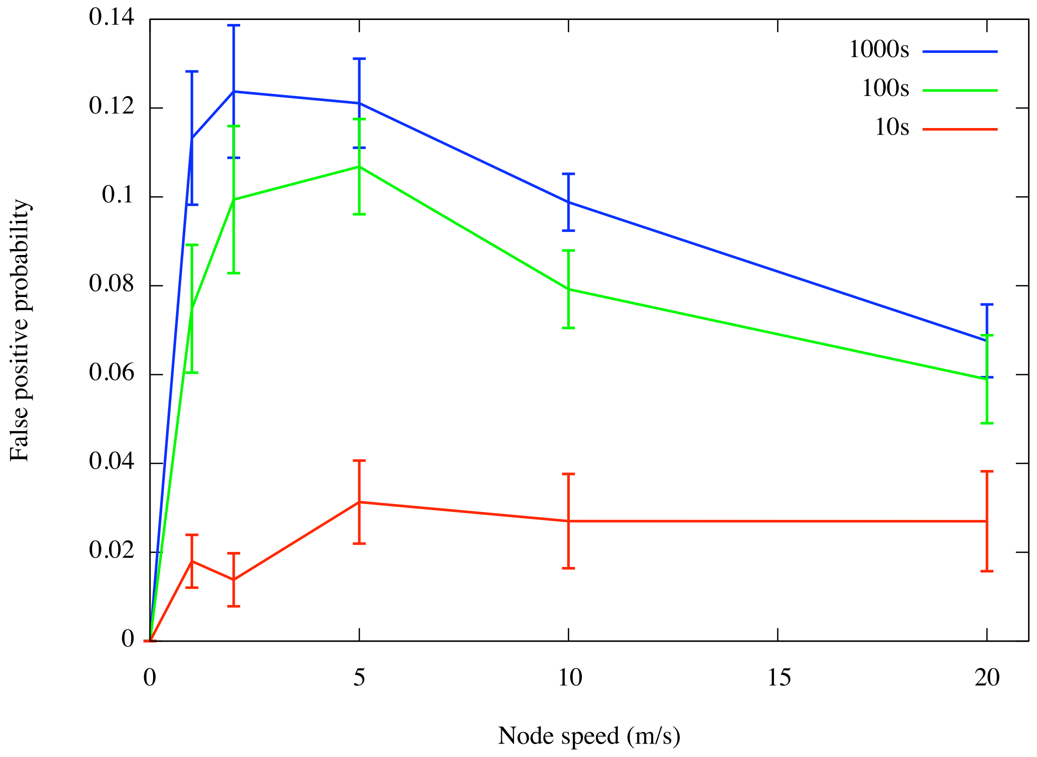


Figure 9.2: False positive probability caused by caching in a dense network.

9.3.1 Description

Two scenarios were created; one dense and one sparse. The dense scenario consisted of 22 nodes in a 250m x 550m area. The sparse scenario increased the area to 500m x 1000m. In both cases, the nodes followed the random waypoint model from [11] with constant speeds (0,1,2,5,10 m/s) and no pause times. Each of the nodes advertised one service, and these services were randomly requested. Each service lookup that was found in the local cache of a node when the service provider was out of reach (not in the routing table), was counted as a false positive. 20 simulations were run for each combination of node speed and cache time and the 95% confidence interval was estimated and presented in the figures.

9.3.2 Results

The results show the false probability using caching. We observe from Figure 9.2 that an application requesting a service has a probability up to 12% of receiving a false positive reply when a cache timeout of 1000s is used. Even with 100s timeout, the probability is above 10%.

By examining the sparse setup, Figure 9.3, we see that the false positive probability increases considerably. A sparse setup is more likely to form network clusters, which in turn yields erroneous cache entries. By foreseeing a realistic network with nodes moving at 2-10m/s and a cache timeout between 50-100s, the figure show that a false probability of more than 50% can be expected.

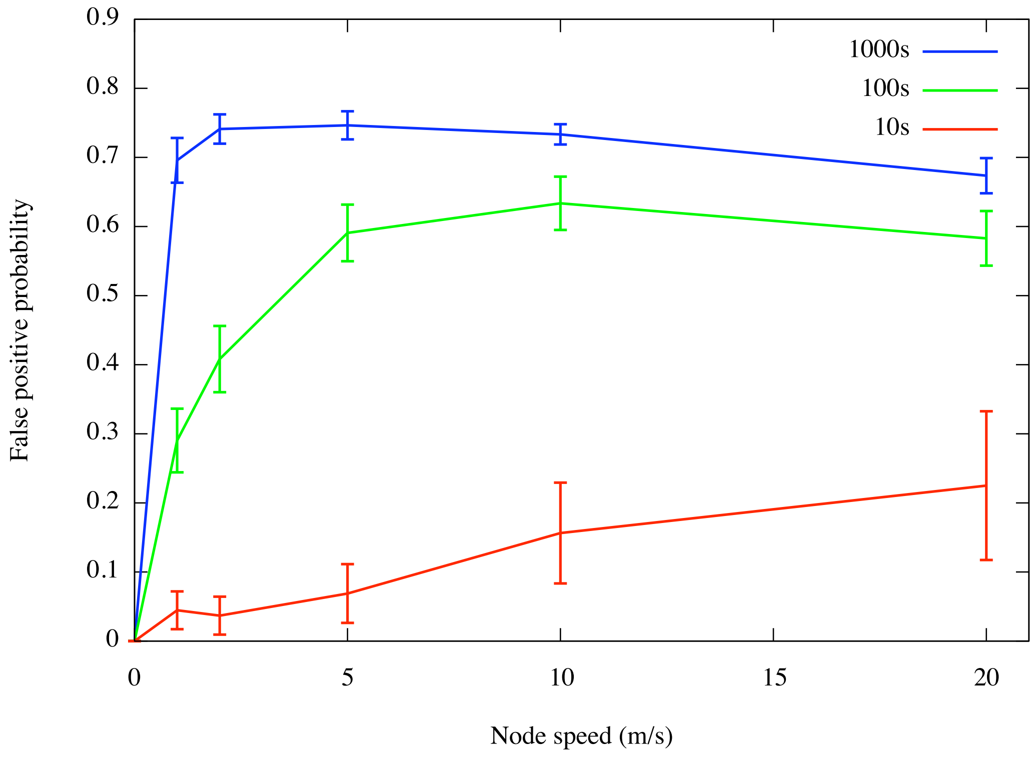


Figure 9.3: False positive probability caused by caching in a sparse network.

In both the dense and the sparse scenario, the false positive probability can be effectively reduced to zero using the path-aware algorithm. The reduction is achieved since the algorithm verifies node availability using the local routing table and initiate a *new* discovery in the network if necessary. Without this cross-layer interaction, false positive replies are inevitable.

An amount (albeit relatively small) of false positive replies may still occur, as the routing table may contain links to nodes out of reach.

Observation

The astute reader may observe that the false positive rate tend to decrease as the node speed increases. I found that the routing tables contained more nodes as the speed increased. This phenomenon could be caused by a combination of different independent factors: (i) The random waypoint model does not distribute the nodes uniformly and they are more likely pass through the center of the simulated area as the walking distance increases. (ii) The combination of the HELLO interval and the transmission range give a higher probability to maintain a OLSR link as the speed increases.

This phenomenon is not fully understood and deserves further examination.

9.3.3 Conclusions

The following conclusions are drawn from the results:

- Caching induce a probability of false positive lookups when service requests are performed. The probability can be as high as 10-50% depending on the node density, node speed and cache timeout.
- The Mercury path aware caching effectively reduces the false positive probability.
- Further work is necessary to examine how OLSR routing parameters affect the validity of the routing table—which in turn affect the performance of the service discovery architecture.

9.4 Comparing Mercury with existing application layer protocols

As described in section 3.1 on page 15, there are two different approaches when designing a service discovery protocol: Either (i) using application-layer service discovery with service dissemination utilizing IP-multicast, or (ii) to use the unicast routing protocol in a cross-layer fashion and perform service dissemination by extending the routing control messages.

As Mercury belongs to the latter category, I wanted to make a qualitative benchmark of the overhead induced by the service discovery process and the average time consumed when requesting a service *compared with* two existing application layer service discovery protocols. PDP [12] and SLPManet [2] were chosen as two independent counterparts in the comparison. Both PDP and SLPManet come with ns-2 code and example simulations.

However, both PDP and SLPManet implementations suffered from limitations and errors, which made it difficult to create a wide range of valid simulation scenarios. SLPManet did not support simulation of both a service provider and a service requester simultaneously on the same node. PDP on the other hand did not handle more than two service providers in the same scenario. An additional problem occurred with the ns-2 scheduler when running the PDP code in certain scenarios. Finally, PDP did not take the length of the service descriptor into consideration when calculating the packet size. This deficiency was corrected.

Both PDP and SLPManet rely on an underlying multicast routing protocol. Notice that multicast in ad-hoc networks is still an open issue (no standard is defined). Simplified Multicast Forwarding (SMF) [61] is, however, proposed by the IETF and represents one of the most promising proposals to solve multicast in MANETs.

Simulating SMF is possible using the nrlolsr [67] implementation for ns-2, and was used for the simulations of PDP and SLPManet. To provide the best working conditions for PDP and SLPManet, SMF was used in S-MPR mode as this is one of the most effective and robust multicast approaches [62]. In contrast to UM-OLSR, nrlolsr did not consider the size of the UDP and IP headers when creating simulation traces. This deficiency was corrected.

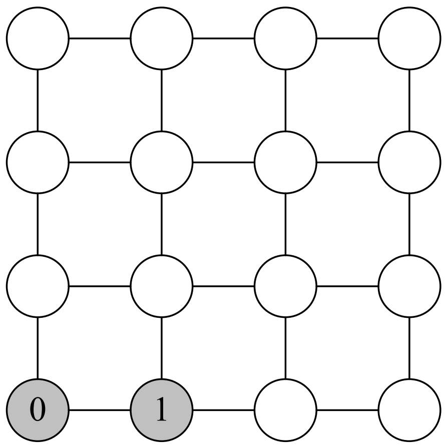


Figure 9.4: Static model used to measure the service discovery overhead.

9.4.1 Measuring overhead

Description

A set of different static topologies were used to measure the overhead. The topologies consisted of nodes oriented in squares of $\{4, 9, 16 \ldots 64\}$ nodes. Figure 9.4 shows the 16-node setup. All topologies had two services, located on node 0 and 1. The services were randomly requested by the other nodes with 5s intervals during the 1500s run. For each static topology, 20 simulations were run and the 95% confidence interval was estimated and presented in the figures. Mercury was configured both without caching in order to reveal the exact discovery overhead, and with 300s caching—a setting more realistic in a final deployment. The service descriptors had a length of 10-15 characters.

Results

Figure 9.5 shows the average network traffic induced by one single service discovery with increasing network size. Compared to its counterparts, the service discovery overhead is reduced by a factor of 20 when using Mercury. The numbers show that Mercury induces less traffic than the two counterparts and that the performance gains are considerable. The overhead reduction using Mercury is partly due to the service descriptor compression achieved from the Bloom filters (compared to transmitting the service descriptors as text), and partly due to the piggybacking of the information in OLSR packets.

9.4.2 Measuring delay

Description

The number of hops between the service request node and the service provider is the factor that has the greatest effect on the service discovery delay. To isolate and measure the time

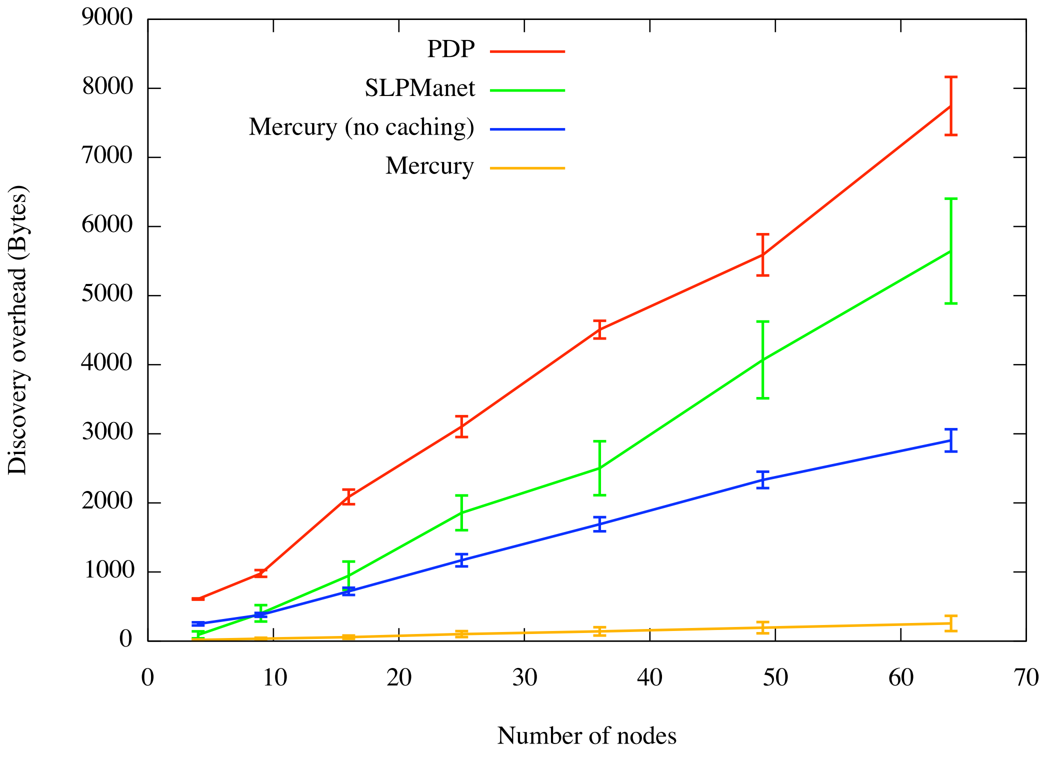


Figure 9.5: Overhead using Mercury compared with SLP and PDP.

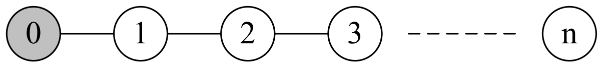


Figure 9.6: Static model used to measure the service discovery delay.

delay, a static network of nodes was chosen. The nodes were connected in chains of 2-16 nodes, yielding 1-15 hops (Figure 9.6). The only service in the network was located on node 0 and was requested by the node in the opposite end of the chain with 10s intervals. The delay between a service request and the successful receipt was measured for 100 requests. In the simulation, both Mercury and SLPManet utilize local caching with 300s timeout. A simulation was also performed with caching switched off (timeout 0s) on Mercury for comparison.

Results

Figure 9.7 show the delay using Mercury (with and without caching) together with SLPManet and PDP. As shown, Mercury without caching is considerably slower than the counterparts. This is caused by OLSR packet forwarding which is slower than IP forwarding. OLSR uses a jitter time in order to support piggybacking of several OLSR packets to one common header. During this jitter time, the packets are delayed. However, considering the results in 8.2.5, I state that using an AODV based service discovery protocol, the service discovery delay would have been increased further.

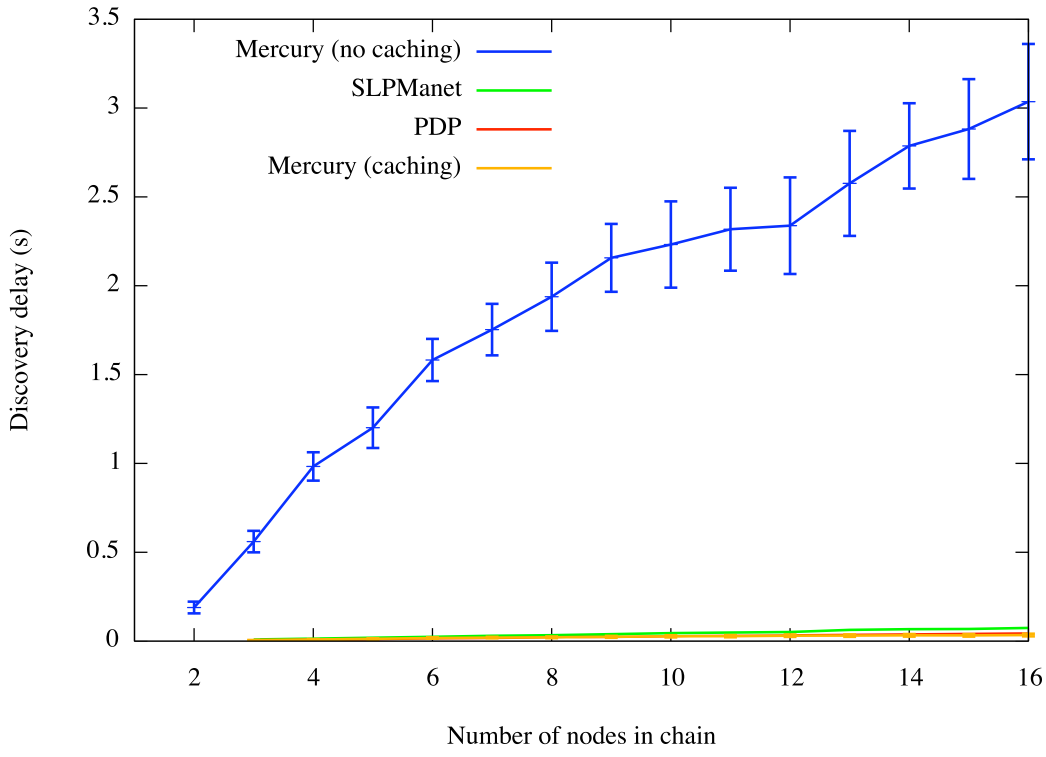


Figure 9.7: The service discovery delay using Mercury and no caching compared with SLP and PDP.

Using a caching timeout of 300s (which is more realistic than 0s in a real-world setting), the result looks promising for all of the three service discovery alternatives (Figure 9.8). In this test, the number of nodes was increased to 20. Notice that Mercury performs better than SLPManet. As both protocols employ caching, they were expected to show equal performance.

Both SLPManet and PDP had delay fluctuations making the estimated the 95% confidence interval to wide to show in the figure. With PDP I measured discovery delays up to several seconds for some node configurations. Most probably, those results were caused by errors in the PDP implementation. Therefore, I chose to omit them from the figures as the paramount intention with this test was to compare application-layer service discovery with cross-layer service discovery and not to compare quality of the protocol implementations.

Notice that the time consumed to connect to the actual service is not considered in this test. This particular time can be many times higher than the discovery delay found in these simulations.

9.4.3 Conclusions

The following conclusions are drawn from the results:

- Thanks to the optimizations included in the Mercury architecture, the service discovery overhead is reduced by a factor of 20 compared to application layer protocols.
- The delay induced in a discovery process is effectively reduced when caching is enabled.

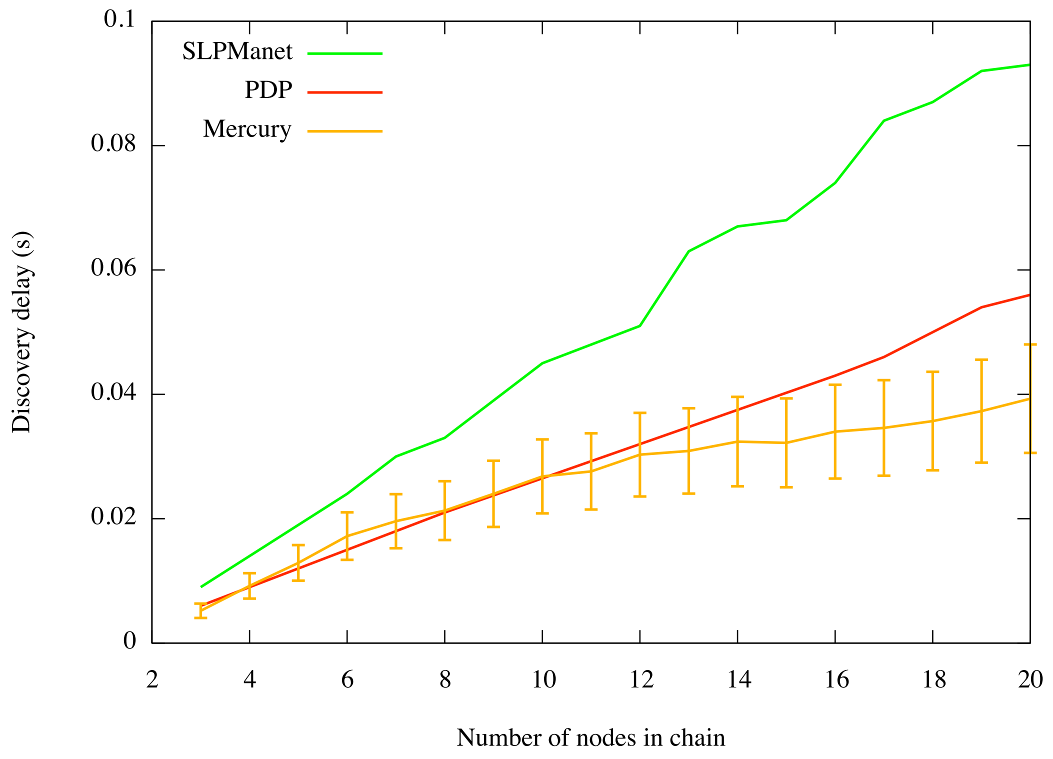


Figure 9.8: The service discovery delay using Mercury with caching compared with SLP and PDP.

With caching, the average delay in a realistic scenario is reduced with more than 90%, and the delay is equal to, or lower than application-layer protocols.

9.5 Comparison of real-world and simulated environment

9.5.1 Description

In order to validate the olsrd-implementation described in chapter 7, a real test was performed with a limited number of nodes. Four laptops were equipped with WLAN cards and olsrd 0.5.5 configured according to the RFC [20]. The nodes were aligned according to Figure 9.6.

In order to compare the exact service discovery delay, caching was turned off in the Mercury plugin. The delay was defined as the time consumed between a service request from an application to a successful reply was received in the same application. 100 such requests were performed, and the 95% confidence interval was estimated and presented in the figures.

9.5.2 Results

Figure 9.9 shows the delay measured in the real test compared to results from the simulation described in 9.4.2 on page 79. As shown, there is a strong correlation between the simulated

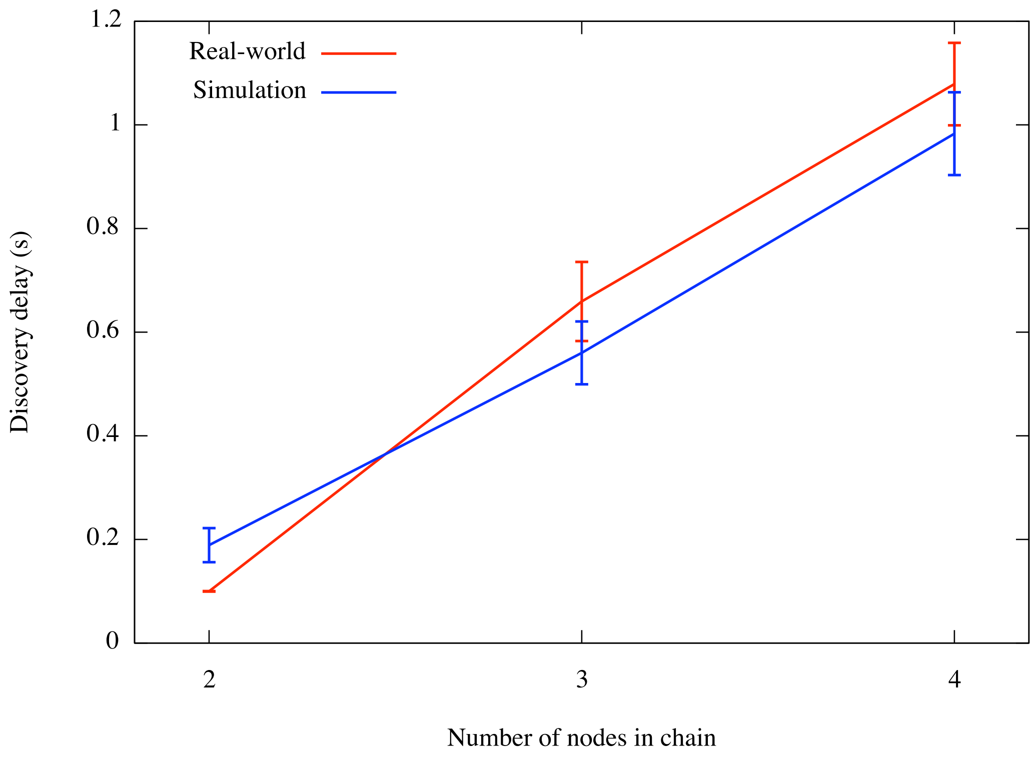


Figure 9.9: Comparing the delay measured by simulation and measured in a real-world test.

results and the real-world measurements.

The variation in the results can be explained by different process priority, other operating system settings and the effect of real radio propagation in the real experiment compared to the simulated environment.

9.5.3 Conclusions

The following conclusions are drawn from the results:

- There is a strong correlation between the simulation and the real-world measurements regarding service discovery delay.
- The implementation of olsrd works as expected by the simulations and is valid for future tests and real-world deployments.
- The service discovery overhead was not measured in the real network. The overhead should correspond with the simulations, however, this is a task for future tests.

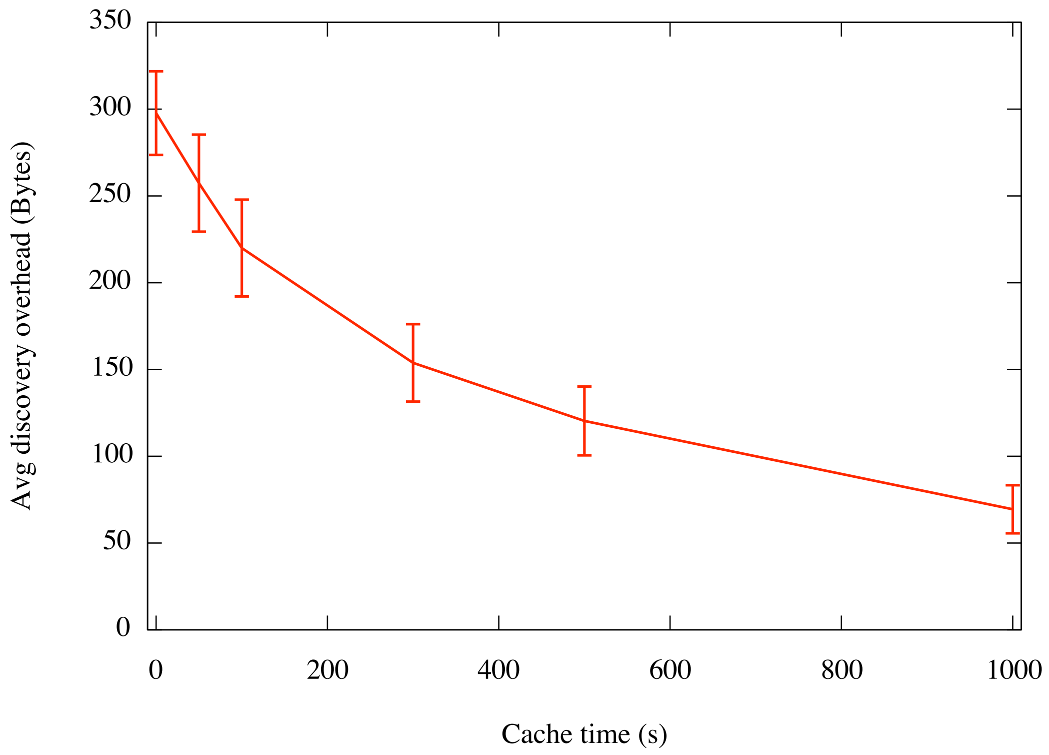


Figure 9.10: Overhead in the real-track simulation with different cache time.

9.6 Performance using real tracks

9.6.1 Description

I wanted to compare the performance of Mercury using synthetic mobility with real mobility tracks gathered from a real-world experiment. The scenario introduced in 8.2.2 on page 66 was used to evaluate Mercury using real tracks. The purpose of the simulation was to give Mercury realistic working conditions.

Every node in the network advertised one service. Services were randomly requested by a random node in 10s intervals during the 1550s simulation. The overhead and the delay induced by each service discovery were then measured with different cache timeouts. The overhead was averaged and the 95% confidence interval was estimated and presented in the figures.

The confidence interval for the delay measurements was not estimated, due to the variance in delay—caused by caching and network clustering.

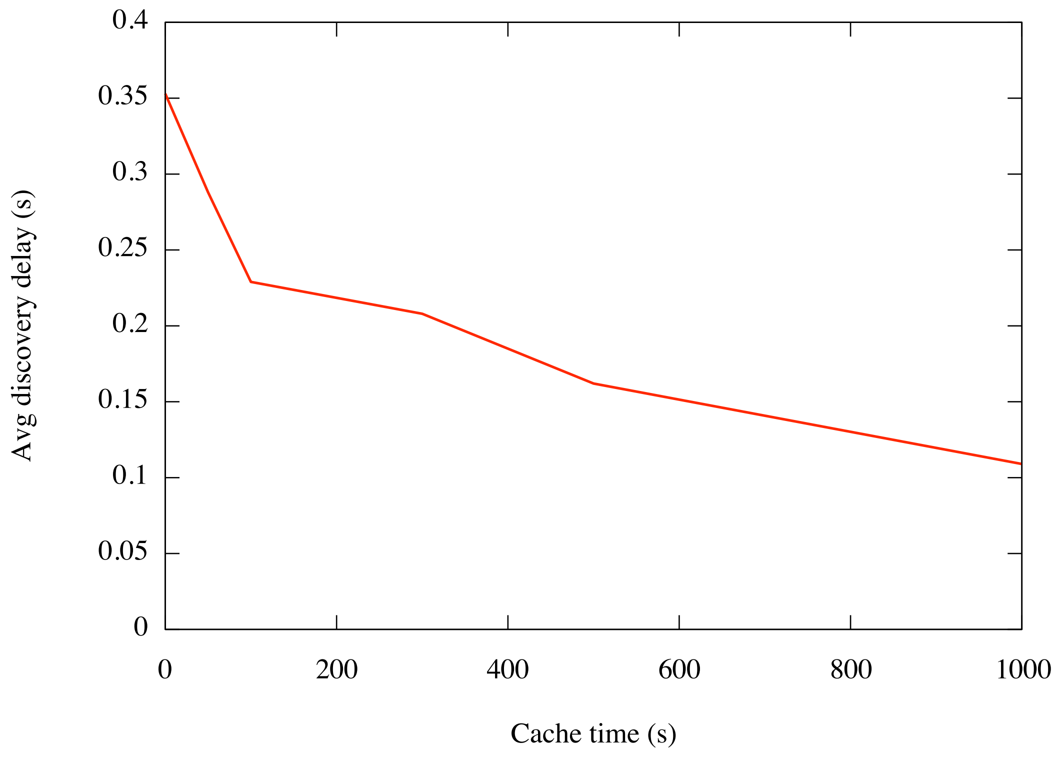


Figure 9.11: Service discovery delay in the real-track simulation with different cache time.

9.6.2 Results

Overhead

The service discovery overhead decreases with increasing cache time (Figure 9.10). The measured overhead correspond to the results found in the static network topology in Figure 9.5 on page 80.

Delay

The service discovery delay decreases with increasing cache time (Figure 9.11). Compared to the delay measured in the static network in Figure 9.8, the delay has increased. This is expected, as the mobile scenario consist of periods of network clustering. A service request will be delayed during the period a service provider is out of reach. In the static scenario, all nodes are available at all time, which explains why the service discovery delay is an order of magnitude larger in the mobile scenario[1].

[1]The nature of movement affect the performance, as discussed in 8.2.

Observation

It is worth noting that in this particular 1550-second scenario, the nodes first move in three independent groups, and then they move together and collaborate. Since the node availability gradually increase during the simulation (see Figure 8.5 on page 70), false positive cache queries are not very likely to occur compared to using random movement—simulated in 9.3. In fact, the number of false positives measured was almost insignificant. A different real-track scenario with more mobility, sparse node distribution, and a more aggressive service discovery pattern may yield a completely different result.

9.6.3 Conclusions

The following conclusions are drawn from the results:

- Caching reduces both the overhead in the discovery process and the average discovery delay.
- Network clustering and mobility increases time consumed in the service discovery process.
- The mobility pattern greatly influences the performance of the service discovery protocol.

Chapter 10

Conclusion

There's two possible outcomes: if the result confirms the hypothesis, then you've made a discovery. If the result is contrary to the hypothesis, then you've made a discovery.

Enrico Fermi

This chapter summarizes the results in the thesis. I also consider my results in relation to other research in the area and presents suggestions for future work.

10.1 Major contributions in the thesis

The aim of this project was to investigate and create a service discovery protocol for bandwidth-constrained environments. One important part of the work was to evaluate the proposal in a realistic setting. The proposal was evaluated by simulation, compared to existing protocols, and implemented for real-life usage.

The major contributions in this thesis are summarized as follows:

- *Design* of a new service discovery protocol (Mercury). The protocol utilizes the OLSR routing protocol in a *cross-layer* fashion and *piggybacks* service advertisements and requests to ordinary routing traffic. The protocol takes advantage of *caching* to reduce unnecessary service requests, and utilizes an optimized way to describe services using *Bloom filters*.
- *Implementation* of Mercury for the *ns-2* network simulator and as a plugin to *olsrd*. The first implementation makes comprehensive simulations possible. The latter implementation is for real-life usage, and makes it possible to use service discovery in any distributed application using a simple interface. An example is demonstrated in appendix C where a SIP user agent has been extended to utilize service discovery.
- *Evaluation* of different scenarios and mobility models for ad-hoc network research. A framework for testing service discovery by static models, synthetic mobility and mobility by using real-tracks is provided.

The protocol and simulation results was presented at the 4th OLSR Interop / Workshop, Canada 2008. The published paper is attached in appendix E. The implementation is made available as open source available for downloading at [27].

10.2 Summary of results

I compared two routing protocols (AODV and OLSR) using two different mobility models: random waypoint and tracks from a real exercise. The results clearly showed that the mobility model had a greater influence on the overall performance in the network than the choice of routing protocol. This served the basis for the different scenarios used in the subsequent simulations.

The Mercury service discovery protocol consists of several components to facilitate service discovery in bandwidth-constrained environments. The false positive probability of the Bloom filter was evaluated by simulation concluding that the chosen algorithm performed close to the theoretical limit.

The Mercury protocol was compared to two application-layer service discovery protocols (SLP-Manet and PDP). The simulation results showed that Mercury is superior to both proposals regarding overhead and that caching is the most important feature to consider when optimizing for bandwidth. Service descriptor compression achieved from the Bloom filters (compared to transmitting the service descriptors as text), and piggybacking of the information in OLSR packets reduces the overhead further. To the best of my knowledge, no other cross-layer service discovery proposals describe service descriptors as Bloom filters. It is therefore expected that Mercury outperforms cross-layer service discovery proposals such as [43, 57].

The experiments also revealed that caching is fundamental to reduce the average service discovery delay. Mercury performed better than, or equal, than its counterparts regarding service discovery delay.

I carried out some experiments to discover any disadvantages by caching. The experiments showed that using random waypoint mobility model, the nodes were prone to create clusters and hence, the entries in the cache were bound to be false with a certain probability. False positive cache queries are inappropriate both from the application and user perspective. An extension to the cache was therefore introduced to verify node availability by checking the routing table prior to the application feedback. An amount of false positive replies may still occur, as network mobility and routing protocol settings may lead to false entries in the routing table.

It should be noted that when a mobility model with gradually increasing connectivity (using real tracks) was used, the chance of getting a false positive cache entry was almost zero. This proves that the choice of mobility model is extremely important when evaluating any component or algorithm in ad-hoc network research.

10.3 Future work

The implementation of the Mercury service discovery protocol is now fully working. During the work of the thesis new ideas to improvements have come to mind: Additional functionality, ideas to new simulations, and other interesting subjects for future research:

10.3.1 Simulations and tests

I consider the protocol to be thoroughly tested and evaluated. I believe that it is particularly important to use real-tracks in simulation studies to evaluate the protocols in a realistic scenario. The gathering, examination and conversion of real-tracks are, however, time consuming. For this reason, only one single 30-minute track was used in the real-track simulations. Using additional simulations would have been beneficial. Nevertheless, when comparing to random waypoint, the results are conclusive: it is paramount to obtain a *valid* mobility model. For future research of service discovery (or any ad-hoc network protocol), I encourage to take advantage of a huge set of realistic real-track scenarios when performing simulations.

The real-world tests performed in the thesis was limited. The results was, however, concurrent with simulations. A natural step forward is to include Mercury on OLSR enabled low-bandwidth UHF radios [4] for further real-world tests.

10.3.2 Implementation

The following elements in the implementation deserve future work:

- Currently Mercury has no IPv6 support. To add IPv6 support is rather straightforward, as *olsrd* supports both versions.
- The protocol should be extended to include OLSRv2 support. This can be done by creating the Mercury Service Discovery Message as a Type-Length-Value structure (TLV), a part of the generalized MANET Packet/Message Format [19].
- In chapter 4, MD5 was proposed as a hash function. The Mercury-plugin, however, uses a slightly faster hash function that can be easily be replaced with MD5 in the future if desirable.
- The service withdrawal scheme forces each application to withdraw services prior to shutdown. In a future version, the Mercury plugin could handle this per IPC socket basis, and perform withdrawal automatically when an application disconnects.
- IP-autoconfiguration is an additional method to provide auto-configurated MANETs and is not considered in this thesis. Solutions such as [18] could easily be combined with Mercury to provide fully auto-configured MANETs.

10.3.3 Performance optimization

If desirable, the performance of the protocol could be further optimized by employing the following proposals:

- As the mobility model has paramount effect on the performance of the routing protocol, the OLSR parameters should be tuned to fit the target scenario as emphasized in [39, 31].
- A way to reduce the OLSR overhead is to include kernel support of OLSR in order to omit UDP as transmission protocol.
- The use of IP header compression is a possible step to obtain further performance gains, either by using ROHC [9] or MANET tailor made solutions [46].

10.3.4 Interoperability

Interoperability between different service discovery architectures needs to be addressed. The survey [44] addresses service discovery protocols for different military operational levels and discusses how to obtain interoperability between those levels. With Mercury it is possible to summarize an entire service directory (e.g. of XML services) into one single message. However, it is difficult to convert from Mercury Bloom filters to any other non-Bloom filter based service discovery protocol.

One could consider to create a compatibility level between Mercury and other service discovery protocols such as DNS-SD [15] or SSDP [30]. This will allow existing unmodified applications to utilize Mercury in the ad-hoc network.

10.4 Conclusion

Service discovery is one of the most important techniques to lower the user interaction to a minimum and to assist software developers creating user-friendly and well-designed applications for mobile ad-hoc networks.

The successful implementation of Mercury shows that service discovery in mobile ad-hoc networks is feasible. I state that a combination of optimization techniques as presented by Mercury is inevitable in order to support efficient service discovery in bandwidth-constrained environments.

10.4.1 Final remarks

In 2003 Chlamtac et al. stated that no MANET killer application had yet emerged [17]. While this statement may still hold for commercial networks, there are two areas where MANETs are now considered inevitable: First-responder networks (emergency responce) and tactical networks. There has been a major focus during the recent years—both within academia and by the industry—to solve issues to provide reliable ad-hoc network capabilities in those two areas.

Ad hoc technology has now proved to be a very useful tool for meeting the tactical battlefield communication requirements [80]. The industry is now embracing this technology, and in the recent years, several vendors have provided handheld radios with MANET capability.

Mobile ad-hoc networks will continue to evolve and new target applications will probably emerge. I expect that *service discovery* will play an important role in fulfilling the expectations of the future mobile ad-hoc networks.

Appendix A

Bloom Filters

A.1 False positive calculation

Given that m is the length in bits of the Bloom filter, n is the number of service descriptors inserted in the filter, and k is the number of hash functions used, the false positive probability can be calculated as shown in this section.

Presumed that the hash function calculates array positions as a uniform distribution, the probability that a given bit in the filter is not set to 1 is:

$$1 - \frac{1}{m}$$

Hence, the probability that *none* of the hash functions $h_1, h_2, \dots h_k$ has set the given bit in the filter to 1 is:

$$\left(1 - \frac{1}{m}\right)^k$$

If we continue by inserting n service descriptors, the probability that a given bit in the filter m is not set is given by :

$$\left(1 - \frac{1}{m}\right)^{kn}$$

The probability for a certain bit in m is set to 1 is:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Each of the k array positions computed by the hash functions is 1 with the above probability. The probability that the algorithm erroneously claims that a service descriptor is in the set is equal to the probability that *all* of the bits set by the k hash functions is 1:

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{A.1}$$

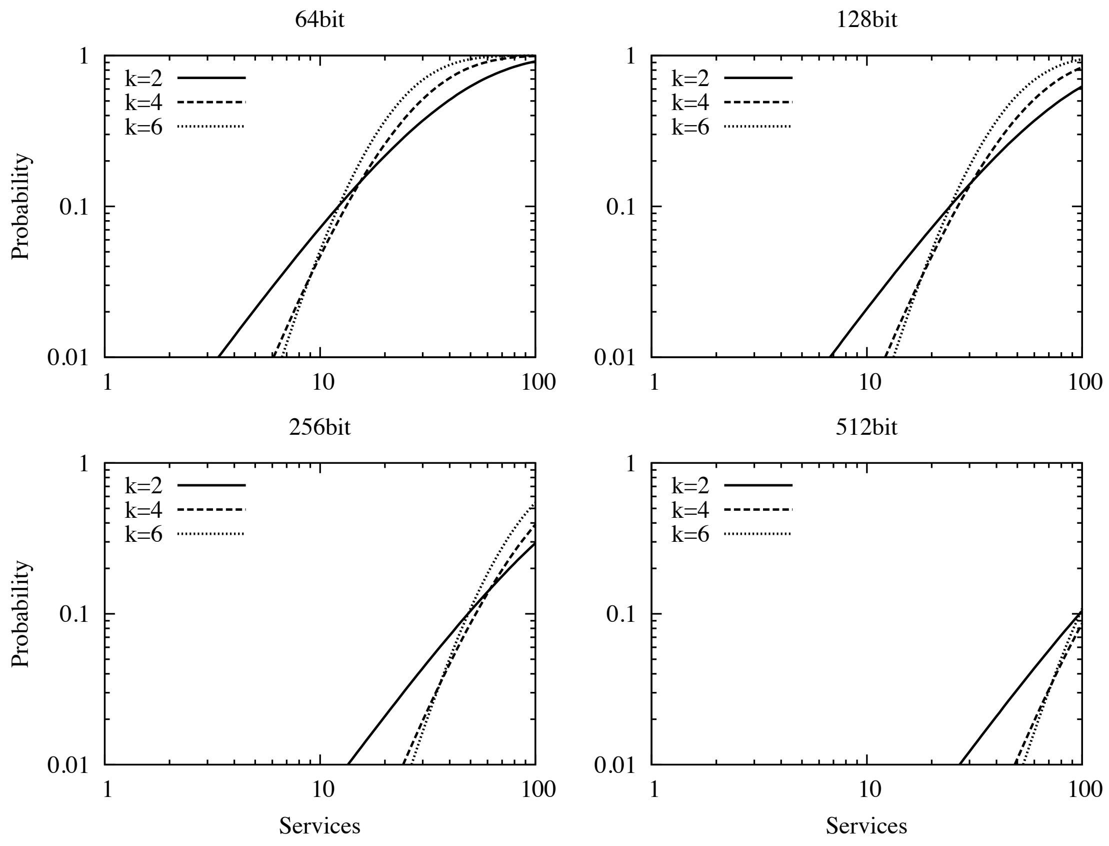


Figure A.1: The false positive probability varies as a the number of hash functions (k) changes, and decreases as the filter size increases

A.2 The optimal number of hash functions

For a given m and n, the optimal number of hash functions k can be calculated by taking the derivate of the equation A.1. If we let $f = (1 - e^{-\frac{kn}{m}})^k$, we find that minimizing the false positive rate is equivalent to minimize df with respect to k:

$$\frac{df}{dk} = \left(1 - e^{-\frac{kn}{m}}\right)^k \left(\frac{k\,n}{m} \frac{e^{-\frac{kn}{m}}}{\left(1 - e^{-\frac{kn}{m}}\right)} + \ln\left(1 - e^{-\frac{kn}{m}}\right)\right) \tag{A.2}$$

Solving equation A.2 with respect to k, we find that the derivate is 0 when $k = \frac{m}{n}\ln 2$. Hence the optimal number of hash functions, k_{opt}, for a filter of width m and a certain number of service descriptors n is then:

$$k = \frac{m}{n}\ln 2 \quad \Rightarrow \quad k_{opt} = \lfloor k \rceil \tag{A.3}$$

A.3 Finding the optimal parameters

By using equation A.1 on page 93, the effect by changing the number of hash functions k and by changing the filter size m can be effectively demonstrated (Figure A.1 on the preceding page). In the Figure, $m = \{64, 128, 256, 512\}$ respectively, and $k = \{2, 4, 6\}$ for all variations of m. Not surprisingly, increasing the size of the filter decreases the false positive probability.

When an appropriate m is chosen that best suits the target application, the optimal value of k can be chosen by estimate the maximum number of services to be held in the Bloom filter, and then use the equation A.3.

In Mercury the default values are $m = 128$ and $k = 4$.

A.4 MD5 in Bloom filters

The actual number of bits that can be set by MD5 is bound by $min(m, 2^{\lfloor \frac{128}{k} \rfloor})$. It is therefore meaningless to increase the size of m above $2^{\lfloor \frac{128}{k} \rfloor}$.

Solving the equation A.1 with respect to n, the maximum number of services that can be stored without increasing the false probability above p can be found as:

$$n \leq \frac{\ln(1 - p^{\frac{1}{k}})}{k \ln(1 - \frac{1}{m})} \tag{A.4}$$

Thus, the maximum number of services that can be stored in a MD5 based Bloom filter using the maximum size $2^{\frac{128}{k}}$ is:

$$n_{max} \leq \frac{\ln(1 - p^{\frac{1}{k}})}{k \ln(1 - 2^{\lfloor \frac{-128}{k} \rfloor})} \tag{A.5}$$

In our context—distributed service discovery in ad-hoc networks—the number of services advertised by each node is certainly not indefinite, and the theoretical upper limit using MD5 is likely never to be reached. Even if we define *a service* to be a fine-grained definition of a utility, application or resource, the total number of services is probably fewer than 20-30 for each node. In this thesis, I therefore claim that in distributed service discovery, a filter size of $2^{\frac{128}{k}}$ is sufficient and that 128-bit MD5 is a perfect match to create Bloom filter hash functions.

Appendix B

Simple simulation example

```
# A simple example for service discovery simulation using Mercury
# Arguments : inputmobilityfile outputtracefile

set val(chan)       Channel/WirelessChannel
set val(prop)       Propagation/TwoRayGround
set val(netif)      Phy/WirelessPhy
set val(mac)        Mac/802_11
set val(ifq)        Queue/DropTail/PriQueue
set val(ll)         LL
set val(ant)        Antenna/OmniAntenna
set val(x)              540   ;# X dimension of the topography
set val(y)              250   ;# Y dimension of the topography
set val(ifqlen)         50           ;# max packet in ifq
set val(seed)           0.0
set val(adhocRouting)   OLSR
set val(nn)             3            ;# how many nodes are simulated
set val(cp)             "cbr-3-test"
set val(sc)             "scenario"

Agent/OLSR set sd_proactive_ false
Agent/OLSR set sd_ival_ 10
Agent/OLSR set sd_cache_ 300
Agent/OLSR set sd_numhash_ 4

# Arguments from command line
# Trace file
set trace [lindex $argv 0]

set val(stop)           200.0          ;# simulation time

Antenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 1.0
Antenna/OmniAntenna set Gr_ 1.0

Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 1.559e-11
Phy/WirelessPhy set RXThresh_ 1.42681e-08 ; #100m
```

```
Phy/WirelessPhy set Pt_ 0.28183815
Phy/WirelessPhy set freq_ 914e+6
Phy/WirelessPhy set L_ 1.0

Mac/802_11 set dataRate_ 1e6
Mac/802_11 set basicRate_ 1e6

# ======================================================================
# Main Program
# =======================================================================
set ns_ [new Simulator]
set topo [new Topography]

set tracefd [open $trace w]

$ns_ trace-all $tracefd
$topo load_flatgrid $val(x) $val(y)
set god_ [create-god $val(nn)]

#global node setting
$ns_ node-config -adhocRouting $val(adhocRouting) \
                 -llType $val(ll) \
                 -macType $val(mac) \
                 -ifqType $val(ifq) \
                 -ifqLen $val(ifqlen) \
                 -antType $val(ant) \
                 -propType $val(prop) \
                 -phyType $val(netif) \
                 -channelType $val(chan) \
 -topoInstance $topo \
 -agentTrace ON \
                 -routerTrace ON \
                 -macTrace OFF

#
#  Create the specified number of nodes [$val(nn)] and "attach" them
#  to the channel.

for {set i 0} {$i < $val(nn) } {incr i} {
set node_($i) [$ns_ node]
$node_($i) random-motion 0
}

$node_(0) set X_ 100.00
$node_(0) set Y_ 100.00
$node_(0) set Z_ 0.0000
$node_(1) set X_ 180.00
$node_(1) set Y_ 100.00
$node_(1) set Z_ 0.0000
$node_(2) set X_ 260.00
$node_(2) set Y_ 100.00
$node_(2) set Z_ 0.0000

# Service access
$ns_ at 1.0 "[$node_(0) agent 255] SD_ADD_SERVICE temp-sensor"
$ns_ at 2.0 "[$node_(1) agent 255] SD_ADD_SERVICE temp-sensor"
$ns_ at 10.0 "[$node_(1) agent 255] SD_ADD_SERVICE IR-sensor"
$ns_ at 20.0 "[$node_(2) agent 255] SD_REQUEST_SERVICE temp-sensor"
```

```
$ns_ at 20.1 "[$node_(2) agent 255] SD_REQUEST_SERVICE IR-sensor"


# Prints service access in trace file
$ns_ at 1.0 "puts $tracefd \"sd 1.0 0 SD_ADD_SERVICE temp-sensor\""
$ns_ at 2.0 "puts $tracefd \"sd 2.0 1 SD_ADD_SERVICE temp-sensor\""
$ns_ at 10.0 "puts $tracefd \"sd 10.0 1 SD_ADD_SERVICE IR-sensor\""
$ns_ at 20.0 "puts $tracefd \"sd 20.0 _2_ SD_REQUEST_SERVICE temp-sensor\""
$ns_ at 20.1 "puts $tracefd \"sd 20.1 _2_ SD_REQUEST_SERVICE IR-sensor\""



for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(stop).0 "$node_($i) reset";
}

for {set i 1} {$i < $val(stop) } {set i [expr {$i + 10}]} {
set percent [expr {$i * 100 / $val(stop)}]
set percent [expr {ceil($percent)}]
$ns_ at $i.0 "puts \"$percent % done\"" ;
}

$ns_ at  $val(stop).0002 "puts \"NS EXITING...\" ; $ns_ halt"

puts "Starting Simulation..."
$ns_ run
```

Appendix C

A real-world test: Discovery of SIP User Agents

In chapter 7, the Mercury service discovery plugin for olsrd was described. In this appendix, I will demonstrate by example how to extend an existing application to utilize service discovery.

C.1 SIP

C.1.1 Introduction

Session Initiation Protocol (SIP) [79] is designed to provide signaling support for multimedia application sessions such as IP telephony, video conferencing and instant messaging. SIP itself is used primarily to set up and tear down multimedia *sessions*, while the multimedia communication itself is usually done over separate protocols such as RTP [84]. In order to negotiate which IP ports to setup and which codes to use, a third protocol—Session Description Protocol (SDP) [35]—is used. These three protocols are usually combined in a SIP enabled multimedia application.

SIP defines several network elements: The *end user element* is called a SIP User Agent (UA) which may in turn connect to a *server element* (proxy, registrar or redirect server) or directly to an other UA.

C.1.2 SIP in MANETs

In fixed networks, centralized Domain Name Service (DNS) servers can be used to locate SIP servers. Such DNS servers do not, however, exist in most MANETs, and further, the binding to a centralized SIP server may represent a single point of failure due to mobility and unstable links. The recommended approach for MANETs is therefore a server-less SIP infrastructure, as illustrated by figure C.1. In order to create such a server less infrastructure, the SIP UAs must connect directly to each other without any server element in between.

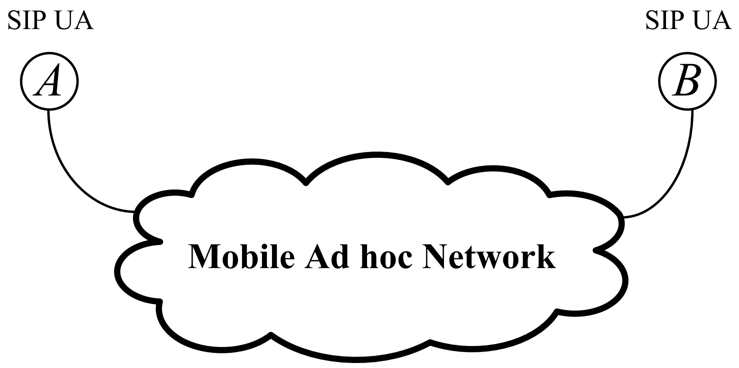


Figure C.1: It is a key issue to determine the location of the SIP User Agents in a Mobile Ad Hoc Network

Previous studies have shown that the challenging problem of finding the location of UAs can be addressed using Service Discovery. Banerjee et al. [5] compare a SIP discovery process *independent* from the routing procedure with a discovery process *integrated* with the routing. The latter approach turns out to outperform the independent approach when it comes to both control overhead and latency in the SIP session setup. Li et al. [58] propose to integrate the SIP discovery process with the OLSR routing protocol, and demonstrate promising results by simulation.

The next section will show a similar approach, where an open source SIP UA is extended to take advantage of Mercury service discovery.

C.2 Code extension

Peers [65] is a minimal SIP user agent written in Java. It enables Voice over IP services by allowing a user to call another user on a Local Area Network or a MANET using SIP/SDP/RTP. Using the default Peers installation, the caller is required to enter the IP address belonging to the node that it wants to call.

In this section I will demonstrate how the Peers user agent can be extended to utilize Mercury service discovery. Using only a few modifications, the application can utilize the service discovery layer to detect the IP address of other SIP enabled nodes: i.e. there is no need to enter the IP address manually.

Using the Inter Process Communication Interface as specified in 7.4.6 on page 59, only a few code lines are necessary in order to extend the existing Java code:

C.2.1 Connect to the plugin

At application startup, the Mercury plugin is connected:

```
mercurySocket = new Socket("localhost",8888);
out = new PrintWriter(mercurySocket.getOutputStream(), true);
```

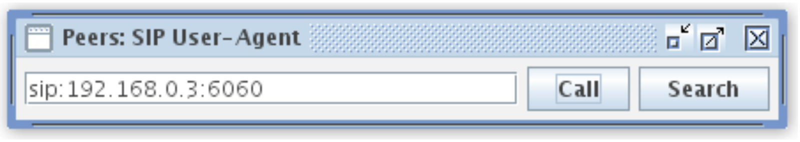


Figure C.2: Mercury has discovered an other SIP User Agent at the address 192.168.0.3. The search is done automatically at startup, but can be initiated manually by pressing the new button "Search".

```
in = new BufferedReader(new InputStreamReader(mercurySocket.getInputStream()));
```

We see that the socket to the Mercury plugin is first established. Then, two objects are created in order *read from* and *write to* the socket.

C.2.2 Advertise the SIP service

After the successful initialization, the service "SIP" is advertised to inform other SIP-clients in the ad hoc network about the existence of the UA:

```
out.println("ADVR SIP");
```

C.2.3 Request SIP services

The application requests for other SIP UAs using a simple command:

```
out.println("RQST SIP ALL");
```

In this setup, the application asks for *all* services of the type "SIP" using the attribute `ALL`. This attribute tells Mercury to retrieve every one of the SIP-services in the network. In contrast, the attribute `ANY` asks for the *first* service—no matter which one. The latter variant may be useful for other purposes.

C.2.4 Parse the plugin output

When the plugin is connected, and the service "SIP" is requested, the client will start to receive IP addresses of the other entire SIP enabled clients (immediately as they connect) via the IPC Interface. A string tokenizer parses the incoming string, adds the chosen SIP port (6060 in the example) and hands the address to the graphical user interface. The look of the interface after the successful discovery of an other UA is illustrated by Figure C.2.

```
String inLine = in.readLine();
if(inLine.startsWith("SERVICE FOUND")){
  StringTokenizer tok = new StringTokenizer(inLine);
  String s1 = tok.nextToken(); // SERVICE
  String s2 = tok.nextToken(); // FOUND
  String s3 = tok.nextToken(); // SIP
  String s4 = tok.nextToken(); // AT
  String s5 = tok.nextToken(); // IP-address

  if(s3.equals("SIP")){
    String SIPuri = "sip:" + s5 + ":6060";
    myGui.updateUri(SIPuri);
  }
}
```

C.3 Summary

This chapter has demonstrated an example of one relevant target application for Mercury service discovery. By using only a few dozen code-lines, the Peers SIP software is changed to automatically detect another SIP UA in the ad-hoc network. Other existing distributed applications such as file sharing, instant messaging, whiteboard sharing, can use the same technique.

Appendix D

Tools

A set of different programming languages, systems and tools were used while working on the thesis.

C# and *Visual Studio* were used to convert ns-2 traces from GPS tracks used in chapter 8. The conversion was performed on *Windows XP*. iNSpect [54] was used to visualize the traces.

C++, *STL* and the *GCC* compiler were used to create the service discovery extension for ns-2 described in chapter 6. The simulations described in chapter 8 and chapter 9 were scripted using *TCL* and analyzed using *bash*, *awk*, *sed* and a variety of CLI programs. All simulations were performed on *Mac OS X*.

The *C* programming language and the *GCC* compiler were used on a *Linux* machine to program the service discovery plugin for olsrd in chapter 7.

The SIP client extension described in appendix C was programmed in *Java* using *Eclipse*.

The plots in the thesis were generated using *Gnuplot*, and drawings and diagrams were created in *Visio*. The thesis report was written in *Latex*.

Appendix E

Publications

E.1 Web Services and Service Discovery

FFI-RAPPORT 2008/01064 Norwegian Defence Research Establishment.
F. T. Johnsen, J. Flathagen, T. Gagnes et al. 2008.

Available on www.ffi.no.

E.2 Service Discovery using OLSR and Bloom Filters

Paper presented at the 4th OLSR Interop / Workshop, Ottawa, CA, October 14-16 2008
J. Flathagen and K. Øvsthus.

Attached on the next page.

Service Discovery using OLSR and Bloom Filters

Joakim Flathagen
Norwegian Defence Research Establishment
BX 25, N-2027 Kjeller, Norway
Email: joakim.flathagen@ffi.no

Knut Øvsthus
Bergen University College
BX 7030, N-5020 Bergen, Norway
Email: knut.ovsthus@hib.no

***Abstract*—Automatic discovery of services and resources is a crucial feature to achieve the expected user-friendliness in Mobile Ad-hoc Networks. Due to limited computing power, scarce bandwidth, high mobility and the lack of a central coordinating entity, service discovery in these networks is a challenging task.**

In this paper, we develop a service discovery protocol (Mercury) utilizing a combination of different optimization techniques: The performance is increased using cross-layer interaction between the application layer and the routing layer. The service information is described using Bloom filters and distributed using Optimized Link State Routing (OLSR). A caching regime is implemented to obtain further reductions of both overhead and latency.

The analysis and simulation results show that our service discovery proposal induces very low overhead to OLSR and is superior to application layer solutions. The proposal is implemented as a plugin to the OLSR implementation *olsrd* for real-world deployments.

***Index Terms*—MANET, OLSR, Service discovery**

I. Introduction

A Mobile Ad-hoc NETwork (MANET) is a collection of mobile nodes connected by wireless links able to dynamically form an arbitrary multihop network—without the use of any pre-existing infrastructure. In order to enable communication between any two nodes in such a network, a special routing protocol is employed. The IETF MANET working group mainly considers two routing approaches: *Reactive routing* such as AODV [18] and *Proactive routing* such as OLSR [5].

However, there is a need for a *service discovery protocol* to discover applications, services and resources in the network. There has been much research activity in the field of service discovery by several consortiums, companies and organizations. This research has produced service discovery mainly for fixed local area networks. Examples include Service Location Protocol (SLP) [9], Simple Service Discovery Protocol (SSDP) [8], Jini [20] and DNS Service Discovery (DNS-SD) [4]. However, the overall Internet community has not yet reached a consensus on one particular service discovery protocol. Moreover, none of the above solutions are applicable to MANETs without adaptations as these networks have less computing resources, lower network bandwidth, higher mobility and more heterogeneity.

Service discovery (SD) mechanisms for MANETs are divided in two groups: (1) mechanisms *independent* of the underlying routing protocol, and (2) mechanisms *integrated* with the routing protocol, be it either *reactive* or *proactive*.

Most of the MANET SD proposals belong to the first category and solves the SD at a layer above routing—referred to as *application layer service discovery*. Examples include SLPManet [1], PDP [3] and Konark [10], which all rely on multicast support on the network layer. The performance of such SD protocols is therefore bound to the chosen multicast protocol. Further, multicast in MANETs is still at the research stage (no standard is defined) and is hence an open issue.

A better and more optimized approach is therefore to implement the SD protocol in a *cross-layer* fashion, and exploit the routing layer for efficient dissemination of service control messages. SEDRIAN [16] and the work by Engelstad et al. [6] propose cross-layer service discovery utilizing AODV. Jodra et al. [11] and Lightweight Service Discovery (LSD) [12] are examples of cross-layer service discovery using OLSR.

Differing from previous work on cross layer service discovery based on OLSR, this paper focus to support low-bandwidth environments and investigates an efficient way to describe services using Bloom filters combined with service caching. The analysis and simulation results show that our optimized SD proposal named Mercury, induces very low overhead to OLSR and outperforms application layer SD solutions. The proposal is implemented for real-world deployments [7].

The remainder of this paper is organized as follows: Section II presents Mercury service discovery protocol in detail. Section III describes the real-world implementation. Section IV and V presents and discusses the simulations. Finally, the paper is concluded by section VI.

II. Our Service discovery design

A. Overview

To successfully create service discovery for bandwidth-constrained environments, we envision several combined optimizations. For this purpose, we propose a new SD solution, Mercury. Mercury describes the service descriptors efficiently as *Bloom filters*, performs service dissemination by *piggybacking* service information on OLSR routing messages and utilizes *caching* of service advertisements.

Mercury handles requests and advertisements from two entities: (1) Local applications on the node and (2) foreign nodes through the ad hoc network (Fig. 1). Each node uses a set of repositories to store the information (Fig. 3): **Advertised services** contains the different services offered by the node itself. The services persist in this list until an upper layer application withdraws the service. Advertisements are sent

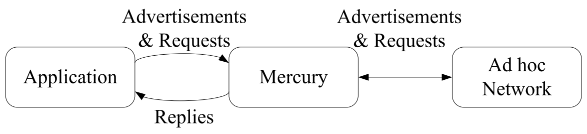


Fig. 1. Mercury connects users and applications to services in the Ad hoc network using service advertisements and service requests.

both when a service is first registered and upon an external request. All the service descriptors in this list are included in advertisemens encoded as one single Bloom Filter. In **Foreign services cache**, all the services offered by other nodes are stored. Each entry in the list consists of the Bloom Filter advertised by the foreign node and its current IP address. The last repository contains the **Requested services** which stores all the services requested—awaiting an incoming advertisement.

All incoming advertisements are immediately stored in the cache. Upon a request from an upper layer application, the cache is first requested. If an entry is found, the application is immediately notified. Otherwise, a service request is sent.

B. Protocol format

OLSR communicates using a unified packet format for all data [5]. Using this format the OLSR standard provides extensibility of the protocol without breaking backwards compatibility. This feature gives a unique possibility to disseminate different kinds of information through intermediate nodes even if the nodes do not support the specific extension.

We take advantage of the extensibility feature of the OLSR format, and introduce a new message, namely the Mercury service discovery message (MSD). MSD messages are sent as the data-portion of the general message format with the message type set to MSD_MESSAGE. The MSD message has the format specified in Fig. 2 when piggybacked to an OLSR header. The Mercury part consists of four fields including a **Spare** field for future use. The **Type** field indicates whether the message is a service request or a service reply. The **Service Filter** field contains the filter describing the services to be requested or advertised encoded as a Bloom filter (described subsequently). The **Filter Length** gives the size of the filter.

C. Distributing service descriptors

Many service discovery protocols use XML to describe the service information, such as in [10]. However, XML requires considerable bandwidth, which is sparse in ad hoc networks. An alternative is to map a predefined set of keywords, or service descriptors, to integers to save bandwidth as proposed in [11]. This solution indeed saves bandwidth. However, it is not very flexible nor is it scalable, as it requires maintenance on every node in the network when new service categories are added.

The proposed solution in this paper is therefore to distribute a summary of the available services as a vector described as a *Bloom filter* [2]. A Bloom filter is a data structure that allows data representation in a simple and space-efficient manner.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Packet Length         |    Packet Sequence Number     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  MSD_MESSAGE  |     Vtime     |         Message Size          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Originator Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time To Live |   Hop Count   |    Message Sequence Number    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      | Filter Length |             Spare             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
:                         Service Filter                        :
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 2. Mercury service discovery format as an extension to the OLSR message format [5]

The filter is created by hashing service descriptors to a size-defined bit array. The size limitation may cause the filter to indicate that a service descriptor is in the filter even though it is not—referred to as a *false positive*. The implementation of the Bloom filter is hence a trade off between the size of the filter and the probability of a false positive request to the filter. Our Bloom filter is implemented using k independent hash functions to hash each service descriptor to the array. Given the number of service descriptors n and the filter width m, the probability of a false positive lookup can be given as:

$$P_n = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \qquad (1)$$

In order to minimize the false positive rate, the filter width should mathematically be as large as possible. However, the feasible size is limited by computation time, OLSR packet size and memory consumption. The optimal value of k can be calculated by taking the derivate of equation 1. We then find that the derivate is 0 when $k = \frac{m}{n} \ln 2$, hence yielding the optimal number of hash functions for a given filter width. By having a thorough understanding of the target application the parameters k and m can be set to minimize the probability of false positive. In Mercury, the parameters are adjustable, however the default values are $k = 4$ and $m = 128$.

In Mercury the filter is created using the message digest function MD5 [19]. MD5 is a cryptographic hash function that hashes arbitrary length strings to 128 bits. The k hash functions can then be constructed from k groups of r bits each out of the 128 bit hash. The Bloom filter in Mercury is implemented as shown in algorithm 1.

Algorithm 1 is used both when services are advertised and requested. An example usage of Bloom filter based SD is shown in Fig. 3. Each node advertises two services and employs three hash functions to describe the services. After performing service requests, the descriptors are stored in the local cache of the other nodes. The cache consists of one Bloom filter for each of the cached nodes (i.e. attenuated Bloom filter).

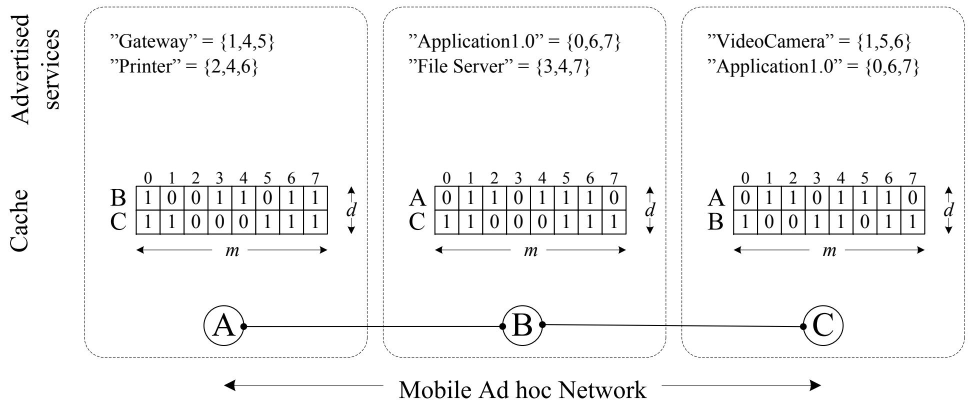


Fig. 3. A Mobile Ad hoc Network consisting of three nodes. Each node use three hash functions to create the Bloom filter and employs two repositories: One repository store the local services advertised, and one repository—implemented as an attenuated Bloom filter of depth d—serves as a cache storing advertisements received from foreign nodes.

Algorithm 1 Calculate the Bloom filter v for service x

```
Require: $x \neq 0$
1: $a \Leftarrow MD5(x)$
2: $r \Leftarrow 128/k$
3: for $i = 0$ to $k$ do
4:     $f \Leftarrow subbits(r * i, (r * (i + 1)) - 1, a)$
5:     $v[f \mod m] = 1$
6: end for
```

D. Caching

Caching is employed to save network bandwidth. Caching may however, lead to false positive replies to the overlying application (Fig. 1) if the advertised service exists in cache even if the node with the advertised service is—due to network clustering—not available anymore. The cache cleanup timeout is therefore a trade-off between fast service queries and the false positive rate. To reduce the amount of false replies to the application, we propose a path-aware approach that consults the local routing table for the availability of the nodes in the cache. If a service exists is in the cache even if the node is not available, Mercury removes the cache entry and performs a new service discovery in order to find relevant nodes offering a similar service.

III. Implementation and use

The Mercury SD proposal is implemented as an extension to the UniK OLSR implementation (*olsrd*) [17]. Olsrd supports the loading of dynamically loaded libraries for auxiliary functions using a generic plugin interface [21]. Here, the Mercury plugin is briefly described and example usage is given. The code is available at [7] for further reference.

In order to allow communication between the plugin and user applications, a simple Inter-process communication (IPC) function is enabled via TCP/IP. Using IPC, services are *requested*, *advertised*, and *withdrawn* using a set of simple commands. By using Mercury and by adding only a few code lines, any distributed application can be extended to facilitate SD—regardless of programming language.

Peers [14] is a minimal SIP user agent (UA) written in Java. It enables Voice over IP services by allowing a user to call another user in the MANET using SIP. Using standard Peers, the caller is required to enter the IP address belonging to the node which it wants to call. By adding a few code lines, the application can utilize Mercury service discovery to detect the IP address of other SIP UAs automatically.

As shown, first the IPC socket is initialized. Then, two objects are created to communicate with the socket:

```
mySD = new Socket("localhost",port);
out = new PrintWriter(mySD.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
        mySD.getInputStream()));
```

After initialization, the service "SIP" is *advertised* to inform other SIP-clients in the ad-hoc network about the existence of the UA by advertising itself (ADVR):

```
out.println("ADVR SIP");
```

The application then immediately *requests* for all other SIP UAs using the code word RQST:

```
out.println("RQST SIP ALL");
```

The application will now receive the IP addresses of all the other SIP enabled clients—immediately as they connect—via the IPC Interface (`in`). The successful discovery of other clients can then be parsed using a simple string tokenizer. By using only a few code lines, the Peers SIP software is changed to automatically detect other SIP UA in the Ad hoc network. Other existing distributed applications such as file sharing, instant messaging, whiteboard sharing, may use the same technique.

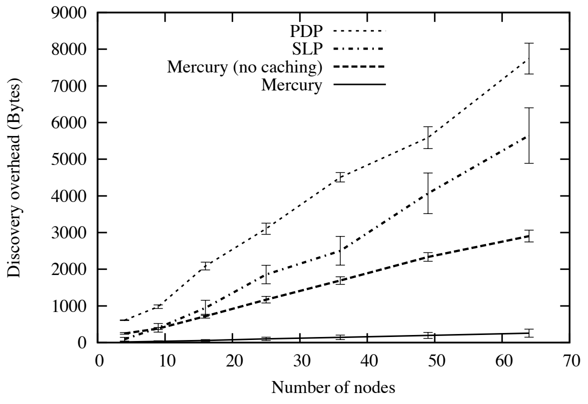


Fig. 4. Overhead using Mercury compared with SLPManet and PDP.

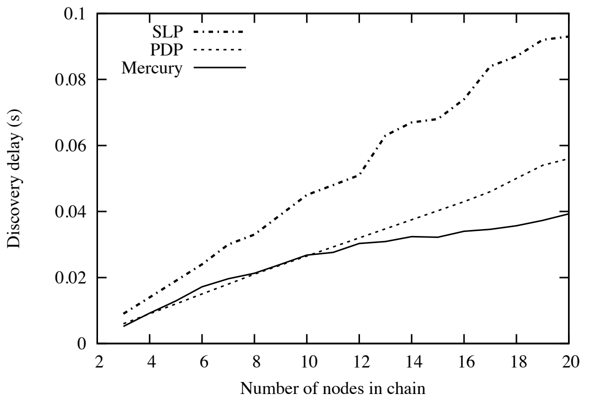


Fig. 5. The service discovery delay using Mercury compared with SLP and PDP.

IV. Performance evaluation

A. Simulation setup

The proposed service discovery mechanism is implemented in ns-2.31 [22] as an extension to UM-OLSR [23]. The transmission range is set to 100m and default OLSR parameters according to [5] are used. For Mercury, the Bloom filter size is set to 128 bits. All measurements are done after topology convergence.

To make a qualitative benchmark of the overhead induced by the service discovery process and the average time consumed when requesting a service, the Mercury protocol is compared with two widespread service discovery protocols, PDP [3] and SLPManet [1]. As both PDP and SLPManet require an underlying multicast routing protocol, our simulation of PDP and SLPManet used *nrlolsr* [15] for ns2 with the extension Simplified Multicast Forwarding (SMF) [13] used in S-MPR mode. Mercury used OLSR MPR message forwarding.

B. Overhead

To measure the overhead, we used static square topologies consisting of 4 to 64 nodes. The network had two services, located on node 0 and 1. The services were randomly requested by the other nodes with 5s intervals during the 1500s run. For each static topology, 20 simulations were run and the 95% confidence interval was estimated and presented in the figures.

Fig. 4 shows average network traffic induced by one single service discovery with increasing network size. Compared to its counterparts, the service discovery overhead is reduced by a factor of 20 when using Mercury.

C. Delay

To measure the time delay when requesting a service, a static network of nodes was chosen, and the nodes were connected in chains of 2 to 20 nodes. The only service in the network was located on node 0 and was requested by the node on the edge of the chain with 10s intervals. The delay between a service request and the successful receipt was measured. Both Mercury and SLPManet utilize local caching with 300s timeout, which reduces the average time delay. The average delay results from all topologies are given in Fig. 5. For all topologies, Mercury performs better or equal than its counterparts.

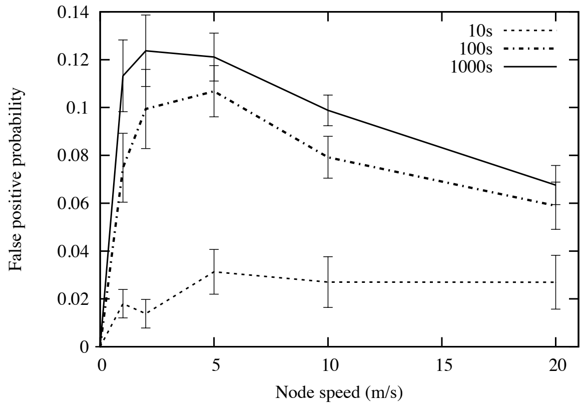


Fig. 6. False positive probability caused by caching in a dense network.

D. Path-aware algorithm

False positive replies as a side effect of caching cause unacceptable delays and reduces user satisfaction. The benefit using our path-aware caching algorithm is clearly showed by the simulations. We created two scenarios, one dense and one sparse. The dense scenario consisted of 22 nodes in a 250m x 550m area. The sparse scenario increased the area to 500m x 1000m. In both cases, the nodes followed the random waypoint model with constant speed. The nodes advertised one service each, which was randomly requested. 20 simulations were run for each combination of node speed and cache time and 95% confidence interval was estimated.

The results show the expected false probability using caching. We observe that an application requesting a service has a probability up to 12% of receiving a false positive reply when a cache timeout of 1000s is used (Fig. 6). The

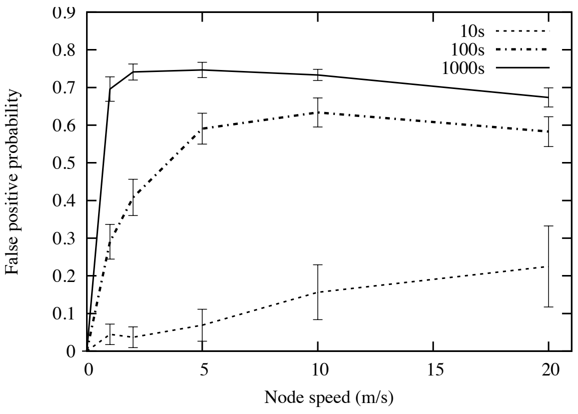


Fig. 7. False positive probability caused by caching in a sparse network.

astute reader may also observe that the false positive rate in some cases tend to decrease as the node speed increases. This phenomenon is caused by general increased node availability (more entries in the routing table) as node availability increase with increasing speed due to the nature of the random way-point mobility model.

By examining the sparse setup (Fig. 7), we see that the false positive probability increases considerably. The false positive rate is effectively reduced using our algorithm since it verifies node availability by examining the routing table.

V. Discussion

The performance results reveal that Mercury is superior to SLPManet and PDP regarding overhead. The major overhead reduction is caused by caching. Service descriptor compression achieved from the Bloom filters (compared to transmitting the service descriptors as text), and piggybacking of the information in OLSR packets further reduce the overhead. Due to these optimizations, it is expected that Mercury outperforms other cross-layer SD proposals [11] and [12].

The time consumed to connect to the actual service is expected to be many times higher than the discovery delay found in the simulations. We therefore state that the service discovery delay is promising for all service discovery alternatives. However, caching is a way to achieve further reduction of the delay.

The proposed path-aware caching architecture, reduces the number of false positives and hence, increases application performance and user-friendliness. An amount (albeit relatively small) of false positive replies may still occur, as network mobility and routing protocol settings may lead to erroneous entries in the routing table.

We state that a combination of optimization techniques as presented by Mercury is inevitable in order to support service discovery in bandwidth-constrained environments.

VI. Conclusions and future work

In this paper, we have presented a method of service discovery using a combination of Bloom filters, the extensibility feature of the OLSR, and a path-aware caching regime. The false positive property of Bloom filters is evaluated and discussed. By simulation, we have demonstrated the performance gain by our cross-layer protocol compared to application layer service discovery alternatives. We also have provided an implementation for real-world usage available for download. Future work includes further optimizations and tests in real deployed networks focusing on bandwidth-constrained environments.

References

[1] M. Abou El Saoud, T. Kunz, and S. Mahmoud. SLPManet: service location protocol for MANET. In *IWCMC '06: Proceeding of the 2006 international conference on Communications and mobile computing*, pages 701–706, New York, NY, USA, 2006.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] C. Campo, C. Garc'ia-Rubio, A. M. Lopez, and F. Almenarez. PDP: a lightweight discovery protocol for local-scope interactions in wireless ad hoc networks. *Comput. Networks*, 50(17):3264–3283, December 2006.

[4] S. Cheshire and M. Krochmal. DNS-Based Service Discovery, August. INTERNET-DRAFT draft-cheshire-dnsext-dns-sd-04.txt, Work in progress, 2006.

[5] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), October 2003.

[6] P. E. Engelstad, Y. Zheng, R. Koodli, and C. E. Perkins. Service discovery architectures for on-demand ad hoc networks. *International Journal of Ad Hoc and Sensor Wireless Networks, Old City Publishing (OCP Science)*, 2(1):27–58, March 2006.

[7] J. Flathagen. Mercury Service Discovery Plugin for OLSRd. (http://olsr-mercury.sourceforge.net), Accessed 2008.

[8] Y. Goland, T. Cai, P. Leach, and Y. Gu. Simple service discovery protocol/1.0. INTERNET-DRAFT draft-cai-ssdp-v1-03.txt, Work in progress, 1999.

[9] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard), June. Updated by RFC 3224, 1999.

[10] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - a service discovery and delivery protocol for ad-hoc networks. *Proceedings of the Third IEEE Conference on Wireless Communication Networks (WCNC), New Orleans*, 2003.

[11] J. L. Jodra, M. Vara, J. M. Cabero, and J. Bagazgoitia. Service discovery mechanism over OLSR for mobile ad-hoc networks. *Advanced Information Networking and Applications, AINA*, 2:534–542, 2006.

[12] L. Li and L. Lamont. A lightweight service discovery mechanism for mobile ad hoc pervasive environment using cross-layer design. *Pervasive Computing and Communications Workshops*, pages 55–59, 2005.

[13] J. Macker. Simplified multicast forwarding for manet. INTERNET-DRAFT draft-ietf-manet-smf-05, Work in progress, 2007.

[14] Martineau, Y. Peers SIP User Agent. (http://peers.sourceforge.net/), Accessed 2008.

[15] Naval Research Laboratory. NRL-OLSR. (http://cs.itd.nrl.navy.mil/), Accessed 2008.

[16] A. Obaid, A. Khir, and H. Mili. A Routing Based Service Discovery Protocol for Ad hoc Networks. In *ICNS '07: Proceedings of the Third International Conference on Networking and Services*, 2007.

[17] olsr.org. The OLSR daemon. (http://www.olsr.org/), Accessed 2008.

[18] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[19] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[20] Sun. Jini. (http://www.jini.org/), Accessed 2008.

[21] A. Tønnesen, A. Hafslund and Ø. Kure. The Unik-OLSR Plugin Library. In *The OLSR Interop and Workshop*, 2004.

[22] University of California. ns2 Network Simulator. (http://www.isi.edu.nsnam/ns/), Accessed 2008.

[23] University of Murcia. UM-OLSR. (http://masimum.dif.um.es/), Accessed 2008.

List of Acronyms

AODV	Ad-hoc On-demand Distance Vector
BSD	Berkeley Software Distribution
CBR	Constant Bit Rate
DNS	Domain Name System
DNS-SD	Domain Name System-based Service Discovery
DSR	Dynamic Source Routing
DYMO	Dynamic MANET On-demand Routing Protocol
FSR	Fisheye State Routing
GPS	Global Positioning System
JLS	Jini Lookup Service
KDE	K Desktop Environment
MANET	Mobile Ad-hoc Network
MD5	Message-Digest Algorithm 5
MPR	Multi Point Relay
MTU	Maximum Transmission Unit
OLSR	Optimized Link State Routing Protocol
OSPF	Open Shortest Path First
OWL	Web Ontology Language
PDP	Pervasive Discovery Protocol
RFC	Request For Comment
RIP	Routing Information Protocol
RMI	Remote Method Invocation
RTP	Real-time Transport Protocol
SBDM	Simple DataBase Manager
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SMF	Simplified Multicast Forwarding
SOAP	Simple Object Access Protocol
SSDP	Simple Service Discovery Protocol
TBRPF	Topology Broadcast Based on Reverse-Path Forwarding
TLV	Type-Length-Value structure
TORA	Temporally-Ordered Routing Algorithm
TTL	Time To Live
UniK	University Graduate Center at Kjeller
UPnP	Universal Plug and Play
UUID	Unique Universal Identifier

WOSPF	Wireless Open Shortest Path First
WSDL	Web Service Description Language

Bibliography

[1] M. Abou El Saoud, T. Kunz, and S. Mahmoud. BENCEManet: An Evaluation Framework for Service Discovery Protocols in MANET. *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, 3:860–865, 2006.

[2] M. Abou El Saoud, T. Kunz, and S. Mahmoud. SLPManet: service location protocol for MANET. In *IWCMC '06: Proceeding of the 2006 international conference on Communications and mobile computing*, pages 701–706, New York, NY, USA, 2006.

[3] J. Ahrenholz, T. Henderson, P. Spagnolo, E. Baccelli, T. Clausen, and P. Jaquet. OSPFv2 Wireless Interface Type. Internet-Draft draft-spagnolo-manet-ospf-wireless-interface-01, Internet Engineering Task Force, May 2004. Work in progress.

[4] V. Arneson, K. Øvsthus, O. I. Bentstuen, and J. Sander. Field trials with IEEE 802.11b-based UHF tactical wideband radio. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, pages 493–498, October 2005.

[5] N. Banerjee, A. Acharya, and S. Das. Enabling SIP-based sessions in ad hoc networks. *Wireless Networks*, 13(4):461–479, August 2007.

[6] G. Bianchi. Performance analysis of the IEEE 802.11 distributed coordination function. *IEEE Journal on Selected Areas in Communications*, 18(3):535–547, 2000.

[7] M. J. Blange, I. P. Karkowski, and B. C. B. Vermeulen. Service discovery in heterogeneous wireless networks. *International Workshop on Wireless Ad-Hoc Networks*, pages 295–299, May-3 June 2004.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L-E. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, and H. Zheng. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC 3095 (Proposed Standard), July 2001. Updated by RFCs 3759, 4815.

[10] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2002.

[11] T. Camp, J. Boleng, and V. Davies. A Survey of Mobility Models for Ad Hoc Network Research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

[12] C. Campo, C. Garc'ia-Rubio, A. M. Lopez, and F. Almenarez. PDP: a lightweight discovery protocol for local-scope interactions in wireless ad hoc networks. *Comput. Networks*, 50(17):3264–3283, December 2006.

[13] I. Chakeres and C. Perkins. Dynamic manet on-demand (dymo) routing, May. INTERNET-DRAFT draft-ietf-manet-dymo-09, Work in progress, 2007.

[14] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005.

[15] S. Cheshire and M. Krochmal. DNS-Based Service Discovery, August. INTERNET-DRAFT draft-cheshire-dnsext-dns-sd-04.txt, Work in progress, 2006.

[16] S. Cheshire and D. Steinberg. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., December 2005.

[17] I. Chlamtac, M. Conti, and J. J. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks*, 1(1):13–64, July 2003.

[18] T. Clausen and E. Baccelli. A simple address autoconfiguration mechanism for OLSR. *IEEE International Symposium on Circuits and Systems, ISCAS*, pages 2971–2974 Vol. 3, May 2005.

[19] T. Clausen, C. Dearlove, J. Dean, and C. Adjih. Generalized manet packet/message format, March. INTERNET-DRAFT draft-ietf-manet-packetbb-16.txt, Work in progress, 2008.

[20] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), October 2003.

[21] T. Clausen, P. Jacquet, and L. Viennot. Comparative study of routing protocols for mobile ad-hoc networks. *In Proceeding of The First Annual Mediterranean Ad Hoc Networking Workshop*, September 2002.

[22] G. Costanzi, R. Lo Cigno, A. Ghittino, and S. Annese. Route Stabilization in Infrastructured Wireless Mesh Networks: an OLSRD Based Solution. *Fifth Annual Conference on Wireless on Demand Network Systems and Services, WONS*, pages 109–115, 2008.

[23] J. Dean. Anycast Routing in OLSR MANETs. Presentation held at the 4th OLSR Interop and Workshop Ottawa, 2008.

[24] F. Delmastro. From pastry to crossroad: Cross-layer ring overlay for ad hoc networks. *Third IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom 2005 Workshops*, pages 60–64, 2005.

[25] P. E. Engelstad and Y. Zheng. Evaluation of service discovery architectures for mobile ad hoc networks. In *WONS '05: Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05)*, pages 2–15, Washington, DC, USA, 2005. IEEE Computer Society.

[26] P. E. Engelstad, Y. Zheng, R. Koodli, and C. E. Perkins. Service discovery architectures for on-demand ad hoc networks. *International Journal of Ad Hoc and Sensor Wireless Networks, Old City Publishing (OCP Science)*, 2(1):27–58, March 2006.

[27] J. Flathagen. Mercury Service Discovery Plugin for OLSRd. (http://olsr-mercury.sourceforge.net), Accessed 2008.

[28] Freifunk.net. olsrexperiment.de. (http://olsrexperiment.de), Accessed 2008.

[29] M. Gerla, X. Hong, and G. Pei. Fisheye State Routing (FSR) for Ad Hoc Networks. Internet-Draft draft-ietf-manet-fsr-03, Internet Engineering Task Force, June 2002. Work in progress.

[30] Y. Goland, T. Cai, P. Leach, and Y. Gu. Simple service discovery protocol/1.0, October. INTERNET-DRAFT draft-cai-ssdp-v1-03.txt, Work in progress, 1999.

[31] C. Gomez, D. Garcia, and J. Paradells. Improving performance of a real ad-hoc network by tuning OLSR parameters. *In Proceedings of the 10th IEEE Symposium on Computers and Communications, ISCC 2005*, pages 16–21, 2005.

[32] R. Gupta, S. Talwar, and D.P. Agrawal. Jini home networking: a step toward pervasive computing. *Computer*, 35(8):34–40, Aug 2002.

[33] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard), June 1999. Updated by RFC 3224.

[34] R. Haarman. Ahoy: A proximity-based discovery protocol. Master's thesis, University of Twente, January 2007.

[35] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

[36] C.L. Hedrick. Routing Information Protocol. RFC 1058 (Historic), June 1988. Updated by RFCs 1388, 1723.

[37] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - a service discovery and delivery protocol for ad-hoc networks. *Proceedings of the Third IEEE Conference on Wireless Communication Networks (WCNC), New Orleans*, 2003.

[38] S Hong, S. Srinivasan, and H. Schulzrinne. Accelerating Service Discovery in Ad-Hoc Zero Configuration Networking. *Global Telecommunications Conference, 2007. GLOBE-COM '07. IEEE*, pages 961–965, Nov. 2007.

[39] Y. Huang, S. N. Bhatti, and D. Parker. Tuning OLSR. *In Proceedings of the 17th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, 2006.

[40] Y. Huang, W. He, K. Nahrstedt, and W. C. Lee. Incident Scene Mobility Analysis. *IEEE Conference on Technologies for Homeland Security*, pages 257–262, 2008.

[41] INRIA. OOLSR. (http://hipercom.inria.fr/OOLSR//), Accessed 2008.

[42] J. Haerri, F. Filali and C. Bonnet. Performance Comparison of AODV and OLSR in VANETs Urban Environments under Realistic Mobility Patterns. *Proceedings of the The Fifth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net 2006)*, pages 176–183, 2006.

[43] J. L. Jodra, M. Vara, J. M. Cabero, and J. Bagazgoitia. Service discovery mechanism over OLSR for mobile ad-hoc networks. *Advanced Information Networking and Applications, AINA*, 2:534–542, 2006.

[44] F. T. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth, T. Hafsøe, J. Halvorsen, N. A. Nordbotten, and M. Skjegstad. Web Services and Service Discovery. FFI/RAPPORT 2008/01064, Norwegian Defence Research Establishment, 2008.

[45] D. Johnson, Y. Hu, and D. Maltz. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. RFC 4728 (Experimental), February 2007.

[46] M. Kaddoura and S. Schneider. SEEHOC: scalable and robust end-to-end header compression techniques for wireless ad hoc networks. *Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004. IEEE*, pages 141–146, 2004.

[47] V. Kawadia and P.R. Kumar. A Cautionary Perspective on Cross Layer Design. *IEEE Wireless Commun.*, 12(1):3–11, Feb 2005.

[48] W. Kiess and M. Mauve. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Netw.*, 5(3):324–339, April 2007.

[49] M. Kim, D. Kotz, and S. Kim. Extracting a Mobility Model from Real User Traces. *Proceedings of the 25th IEEE International Conference on Computer Communications, INFOCOM 2006*, pages 1–13, 2006.

[50] R. Koodli and C. E. Perkins. Service Discovery in On-Demand Ad Hoc Networks. Internet-Draft draft-koodli-manet-servicediscovery-00.txt, Internet Engineering Task Force, October 2002. Work in progress.

[51] J. Kopena, E. Sultanik, Gaurav Naik, I. Howley, M. Peysakhov, V. A. Cicirello, M. Kam, and W. Regli. Service-based computing on MANETs: enabling dynamic interoperability of first responders. *Intelligent Systems, IEEE*, 20(5):17–25, 2005.

[52] M. Kropff, T. Krop, M. Hollick, P. S. Mogre, and R. Steinmetz. A survey on real world and emulation testbeds for mobile ad hoc networks. *Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, TRIDENTCOM 2006*, pages 6 pp.+, 2006.

[53] S. Kurkowski, T. Camp, and M. Colagrosso. MANET simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, October 2005.

[54] S. Kurkowski, T. Camp, N. Mushell, and M. Colagrosso. A Visualization and Analysis Tool for NS-2 Wireless Simulations: iNSpect. *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '05*, pages 503–506, 2005.

[55] Y. Lafon and N. Mitra. SOAP version 1.2 part 0: Primer (second edition). Technical report, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[56] F. Y. Li, L. Vandonif, G. Ziccaf, and S. Zanoli. OLSR Mesh Networks for Broadband Access: Enhancements, Implementation and Deployment. *Proceedings of the 4th IEEE International Conference on Circuits and Systems for Communications, ICCSC 2008*, pages 802–806, 2008.

[57] L. Li and L. Lamont. A lightweight service discovery mechanism for mobile ad hoc pervasive environment using cross-layer design. *Pervasive Computing and Communications Workshops*, pages 55–59, 2005.

[58] L. Li and L. Lamont. Support of multimedia SIP applications in mobile ad hoc networks: service discovery and networking architecture. *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, 6:5 pp.+, 2005.

[59] C. Kevin Liu and D. Booth. Web services description language (WSDL) version 2.0 part 0: Primer. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[60] H. Lundgren, E. Nordstrom, and C. Tschudin. Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In *Proceedings of the 5th ACM international workshop on Wireless mobile multimedia, WOWMOM '02*, pages 49–55, 2002.

[61] J. Macker. Simplified multicast forwarding for manet, June. INTERNET-DRAFT draft-ietf-manet-smf-05, Work in progress, 2007.

[62] J. Macker, I. Downard, J. Dean, and B. Adamson. Evaluation of distributed cover set algorithms in mobile ad hoc network for simplified multicast forwarding. *SIGMOBILE Mob. Comput. Commun. Rev.*, 11(3):1–11, July 2007.

[63] S. Magnuson. Call for help: For first responders, high-tech communications still out of reach. In *National Defence*, pages 36–40, March 2008.

[64] B.S. Manoj and Alexandra Hubenko Baker. Communication challenges in emergency response. *Commun. ACM*, 50(3):51–53, 2007.

[65] Martineau, Y. Peers SIP User Agent. (http://peers.sourceforge.net/), Accessed 2008.

[66] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998.

[67] Naval Research Laboratory. NRL-OLSR. (http://cs.itd.nrl.navy.mil/), Accessed 2008.

[68] D. A. Norman. *The Design of Everyday Things*. Basic Books, September 2002.

[69] A. Obaid, A. Khir, and H. Mili. A Routing Based Service Discovery Protocol for Ad hoc Networks. In *Proceedings of the Third International Conference on Networking and Services, ICNS '07*, 2007.

[70] R. Ogier, F. Templin, and M. Lewis. Topology Dissemination Based on Reverse-Path Forwarding (TBRPF). RFC 3684 (Experimental), February 2004.

[71] olsr.org. The OLSR daemon. (http://www.olsr.org/), Accessed 2008.

[72] OMNeT++. (http://www.omnetpp.org), Accessed 2008.

[73] V. Park and S. Corson. Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification. Internet-Draft draft-ietf-manet-tora-spec-04,, Internet Engineering Task Force, July 2001. Work in progress.

[74] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. RFC 1546 (Informational), November 1993.

[75] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[76] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INFOCOM 2002*, 3:1248–1257 vol.3, 2002.

[77] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[78] F. J. Ros and M. R. Ruiz. Implementing a New Manet Unicasting Routing Protocol in NS2. (University of Murcia), 2004.

[79] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916.

[80] NATO RTO. Awareness of emerging wireless technologies: Ad-hoc and personal area networks standards and emerging technologies. Technical Report RTO-TR-IST-035-AC/323(IST-035)TP/32, NATO RTO, 2007.

[81] F. Sailhan and V. Issarny. Scalable service discovery for manet. *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications, PerCom 2005*, pages 235–244, 2005.

[82] J. M. S. Santana, M. Petrova, and P. Mahonen. UPNP Service Discovery for Heterogeneous Networks. *17th International IEEE Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Sept. 2006.

[83] J. Schneider and T. Kamiya. Efficient XML interchange (EXI) format 1.0. W3C working draft, W3C, March 2008. http://www.w3.org/TR/2008/WD-exi-20080326/.

[84] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[85] V. Srivastava and M. Motani. Cross-layer design: a survey and the road ahead. *Communications Magazine, IEEE*, 43(12):112–119, 2005.

[86] J. A. Stine. Cross-Layer Design of MANETs: The Only Option. *Military Communications Conference, MILCOM 2006*, pages 1–7, 2006.

[87] Sun. Jini. (http://www.jini.org/), Accessed 2008.

[88] The Ohio State University. J-Sim Network simulator. (http://www.j-sim.org), Accessed 2008.

[89] A. Tønnesen. Impementing and extending the Optimized Link State Routing Protocol. Master's thesis, UiO, August 2004.

[90] A. Tønnesen, A. Hafslund and Ø. Kure. The Unik-OLSR Plugin Library. In *Proceedings of the The first OLSR Interop and Workshop*, 2004.

[91] UCLA. GloMoSim. (http://pcl.cs.ucla.edu/projects/glomosim), Accessed 2008.

[92] University of California. ns2 Network Simulator. (http://www.isi.edu.nsnam/ns/), Accessed 2008.

[93] University of Murcia. UM-OLSR. (http://masimum.dif.um.es/), Accessed 2008.

[94] J. Wang and X. Lu. Route recovery based on anycast policy in mobile ad hoc networks. In *Proceedings of International Conference on Communication Technology, ICCT 2003*, volume 2, pages 1262–1265 vol.2, 2003.

[95] Wireless network community Roma. Ninux. (http://ninux.org), Accessed 2008.

[96] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. *Proceedings of INFOCOM 2003*, pages 1312–1321, 2003.

[97] L. Zhang, Z. Shi, and Q. Shen. A Service Discovery Architecture based on Anycast in Pervasive Computing Environments. *Proceedings of the 31st Computer Software and Applications Conference, COMPSAC 2007*, 2:101–108, 2007.

[98] Y. Zheng. Service Discovery in On-Demand Mobile Ad-hoc Networks. Master's thesis, UiO, July 2004.

[99] H. Zimmermann. Availability of Technologies versus Capabilities of Users. In *Proceedings of the 3rd International ISCRAM conference*, pages 66–71, 2006.