Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Florian Süss

Service Discovery and Routing in Mobile Ad Hoc Networks

Master's Thesis
MA-2005-03

November 2004 until April 2005

Advisors: Vincent Lenders, Dr. Martin May
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

A MANET represents a very dynamic network where no fixed infrastructure is available. Every node acts as a router and forwards packets for other devices. In such networks, the communication protocols can not rely on centralized servers / services and must be able to adapt rapidly to the network dynamics. The Communication Systems Group at ETHZ proposed a new communication model for this type of networks that is inspired from the properties of electric fields. In this thesis, a routing protocol based on the proposed model was designed and implemented in C under Linux. Several tools for using and testing the implementation were developed as well. All important network scenarios like node mobility or failover were successfully tested with different hardware and software components. The validation tests show that the new service discovery and routing approach works well in a real environment and that the protocol is implemented correctly.

Kurzfassung

Als MANETs bezeichnet man sehr dynamische Netzwerke in denen keine fixe Infrastruktur zur Verfügung steht. Kommunikationsprotokolle in solchen Netzwerken können nicht auf zentralen Servern / Diensten aufbauen und müssen sich schnell an Veränderungen der Netzwerktopologie anpassen. Die Communication Systems Group der ETHZ hat ein Modell für diese Art von Netzwerken entwickelt, das sich an den Eigenschaften des elektrischen Feldes orientiert. Basierend auf diesem Modell wurde in dieser Arbeit ein Routing Protokoll entworfen und in C auf Linux implementiert. Verschiedene Tools mussten entwickelt werden um das Protokoll zu benutzen und zu testen. Alle wichtigen Szenarien wie Ausfallsicherheit und Mobilität der Knoten wurden mit verschiedenen Hardware und Software Komponenten erfolgreich getestet. Diese Validierungstests zeigen, dass der neue Service Discovery und Routing Ansatz in der realen Umgebung funktioniert und dass das Protokoll korrekt implementiert ist.

Contents

List of Figures

Chapter 1

Introduction

In the past, wireless mobile ad hoc networks have become more and more popular in research. A mobile ad hoc network (MANET) is a temporary network built without the aid of any established infrastructure or centralized administration. If two nodes that are outside their wireless transmission range want to communicate, they use intermediate nodes to forward packets between them. All nodes act both as a host and as a router. Thus, in areas with little or no communication infrastructure at all, users are still able to communicate. Mobile ad hoc networks are characterized as follows:

- **No fixed infrastructure** is available. Every node participating in the network must provide the basic functionality of communication like routing and service discovery due to the absence of centralized servers.
- Configuration and operation has to be **self-organizing** without any human intervention.
- The network is **very dynamic**. Nodes may join and leave the network with high frequency and they are expected to move.

Especially the very dynamic nature of a MANET makes service discovery and routing a challenge. The routing algorithms used in the Internet work well for their original design intention (end-to-end communication between fixed locations), but are unsuitable with MANET characteristics. Additionally, service discovery in the Internet relies on centralized infrastructure that is not available in a MANET.

A new class of algorithms to solve the problem of service discovery and routing in the field of mobile ad hoc networks is needed. Different approaches have been developed, for example the routing protocols proposed by the IETF MANET working group ([7], [8], [9], [10]). They primarily focus on the packet routing and therefore, on the flow of data from one node to another. The primary assumption in such routing protocols is that the destination address is known to the source node before sending the packet. Thus, service discovery must be performed in advance as a separate task. In MANETs, the service provider as well as the client is expected to move. After looking up the network address, the nodes may change the locality in the network, and that can cause different problems with the subsequent packet routing. It is possible that, during the communication process, a better suited service provider joins the network. In this case, the client should notice the new situation and get access to the new service provider. Another problem is that the intermediate nodes, that route the packets between a client and a server, could leave the network or move around. Therefore, the routing algorithm should be flexible to adapt to a continuously changing network topology.

To circumvent these problems, the Computer Engineering and Networks Laboratory [2] proposed a new approach (in the following called *MAgNETic Routing*) in [3] and [4] that we believe better fulfills the communication requirements of mobile hosts in ad hoc networks. The approach eases from the end-to-end communication between two specific devices and introduces a more flexible design. The client specifies a *service type* instead of a network address in data packets. A service type represents a specific kind of service (for example if a device has an external connection to the Internet, it is a service instance of the service type 'Internet Gateway'), provided by one or several devices. Service discovery and packet routing is considered as a single task. The routing is not any more achieved using network addresses but with the service type. The challenge of such a new solution is to select the optimal service provider.

The MAgNETic routing protocol can be viewed as a publish/subscribe mechanism. In the *publish* part, all devices periodically advertise the service types they provide. The advertisements serve to establish a *potential field* in the network similar to the electrical field created by point charges (a service instance represents a positive point charge). Note, that a potential field is created for every service type. In the *subscribe* part, a client sends a request that contains the desired service type. A request is considered as a negative test charge and moves along the highest ascent of the potential field to the closest service instance.

In previous work, the MAgNETic routing protocol was implemented in the network simulator *GloMoSim* [6]. The results of the performance tests with the simulator were very promising. So we decided to implement and test the MAgNETic routing protocol in a real environment. The main tasks of this master's thesis were:

- **MAgNETic routing protocol design.** Work out the protocol specifications according to the published papers. The main objects are the protocol messages and headers. The handling of incoming messages is also part of the protocol design.

- **Implementation.** Deliver a complete implementation of the MAgNETic routing protocol. This includes the development of additional tools and applications for testing and using the protocol functionality.
- **Validation tests.** Validate the implementation with functional tests. Do all relevant tests to prove the functionality of the magnetic routing approach in a real environment as well as the correctness of the implementation.

This report is organized as follows: Chapter 2 describes the functionality of the MAgNETic routing protocol. Chapter 3 gives the detailed protocol specifications with the definition of the protocol messages and the new packet header as well as the message handling at a node. The major concepts of the implementation like program execution in several threads are presented in Chapter 4. In Chapter 5, the results of the validation tests are described. Chapter 6 ends the report with an outlook on future work.

Chapter 2

MAgNETic Protocol Description

This chapter gives a survey of the basic functionality and some major definitions of the MAgNETic service discovery and routing protocol. It is described more detailed in [3] and [4].

2.1 Overview

Every device in the network may provide one or several services, in the following called *service types*. A node providing a service type is a *service instance* of this service type. Each service instance is associated with a *capacity* that denotes the quantity of this service type the device offers. For example, the capacity of the service type '*Internet Gateway*' indicates the link capacity of the providers Internet connection.

In periodic advertisement messages, all devices advertise the service instances they are providing. With these messages, all devices compute a *potential value* for every service type existing in the network. The potential values computed by the different devices build a *potential field* for every service type.

To access a service instance, a *client* specifies a service type in a *query message* that is forwarded using the potential field of the desired service type. Every relaying device forwards the query message to the device in range (*neighbor*) with the highest potential value for the desired service type until the query message reaches a service instance. Hence, the query message is routed along the highest ascent of the potential field to the optimal service instance.

If a query message reaches a service instance of the desired service type, the providing node may answer with a *reply message* that is routed back to the query initiator. To *send data* to a service instance or from the service instance back to the client, the data is attached to a query or reply message (all data traffic is handled within these two message types).

The MAgNETic routing protocol uses the soft-state principle, all states are stored with a timeout value. If a service instance leaves the network, the periodic advertisement messages are not sent anymore and all devices in the network remove this advertisement after a specific amount of time.

2.2 Building Routing Tables

To build up the routing tables, all devices periodically send two types of messages: *advertisement messages* (only if the device provides at least one service type) and *neighbor exchange messages* (sent by all devices). These messages are used to compute a potential value as well as the forwarding 'next hop' device for all service types.

2.2.1 Computing Potential Values

With the service advertisements, a device announces all provided service instances and the corresponding capacities. Advertisement messages are forwarded to all devices in the network. Each device collects all advertisements and computes its potential values for all service types. Therefore, the following *distance function* is used to compute the potential value of a single advertisement:

$$\varphi(c, d) = \frac{c}{d}$$

The operands represent the capacity c of the advertising service instance and the distance d (for example the number of hops) from the service instance.

If a node receives several advertisements announcing the same service type, the potential values of the different advertisements are added up. The resulting formula to compute the potential value for a service type with i advertisement is:

$$\phi(c, d) = \sum_i \varphi_i(c_i, d_i) = \sum_i \frac{c_i}{d_i}$$

A query message is routed along the highest ascent of the potential field for the desired service type to a service instance. To guarantee that each query message reaches a service instance, every service instance represents a local maximum of the potential field. This is achieved by setting the potential value of this service type at the providing device to ∞ if the actual node is a service instance of the considered service type.

Figure 2.1 gives an example how to calculate the potential field of the service type 'Internet Gateway' with three service instances:

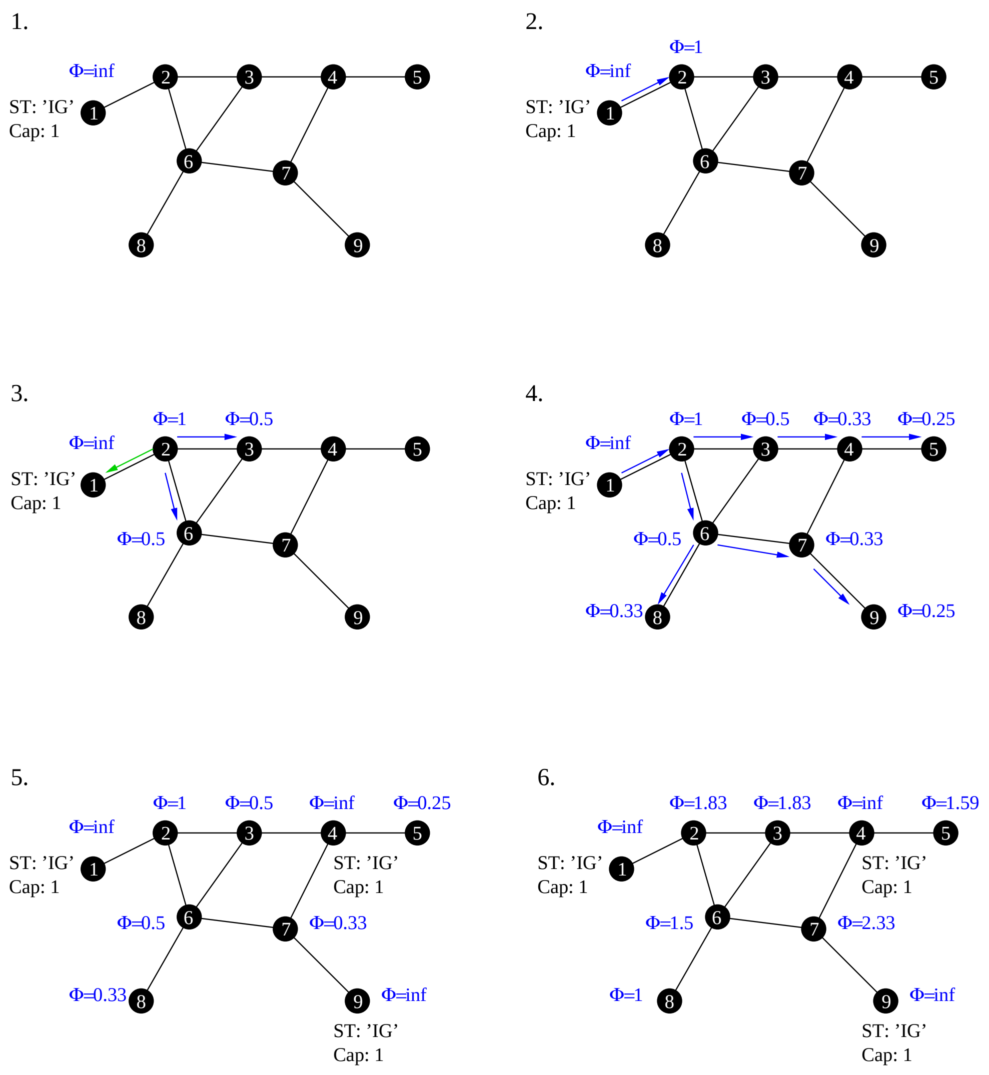

Figure 2.1: Example: Building Potential Fields

1. The example network consists of 9 nodes with the shown connections between them. Node 1 provides an Internet connection with bandwidth 1 Mbit/s, thus it is a service instance for the service type 'Internet Gateway (IG)' and the bandwidth for example can be mapped to the capacity of 1. The potential value ϕ for the service type 'IG' is set to infinite to guarantee a local maximum of the potential field at this node.

2. Device 1 periodically sends advertisement messages announcing the service type 'IG' broadcast to all devices in range. Device 2 receives these messages, saves a copy and computes the potential value (1) for the service type 'IG'.

3. Device 2 forwards the advertisement message to all devices in range, hence it is received by the nodes 1, 3 and 6. Device 1 already knows this advertisement and drops it. Devices 3 and 6 use the received message to compute the potential value (0.5) and forward it again broadcast. Every service advertisement message is distributed this way in the whole network.

4. The potential field for service type 'IG' with the single service instance at node 1 is completely calculated.

5. The devices 4 and 9 also provide the service type 'IG' with capacity of 1 each. They set their potential value for service type 'IG' to infinite and start to send periodic advertisement messages into the network.

6. Complete potential field for service type 'Internet Gateway' and three service instances at nodes 1, 4 and 9.

Such a potential field is built for every service type existing in the network. If a service instance disappears, the potential field is automatically recalculated considering the remaining service instances. This takes some time until the stored advertisement times out and is removed.

2.2.2 Set Forwarding Device

The neighbor exchange messages, that contain the local potential values of all service types are periodically sent broadcast to all devices in range. Every device stores all neighbors and their potential values and sets the neighbor with the highest potential value as forwarding device for every service type in its routing table. Figure 2.2 gives an example how the forwarding device is determined:

1. Consider the routing table of device 6. It contains the entries service type (ST) and the corresponding potential value (P). To compute the

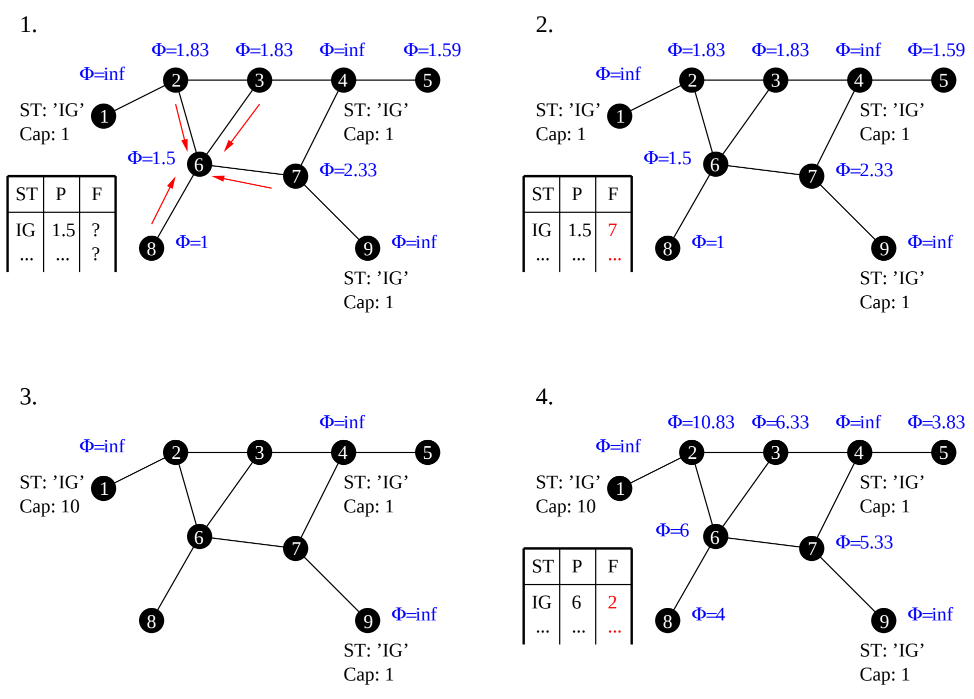

Figure 2.2: Example: Set Next Hop Device

forwarding device (F) for every service type, all devices periodically send neighbor exchange messages. Device 6 receives these messages (from devices 2, 3, 7 and 8) and stores all neighbors and their potential values for all service types into a neighbor list.

2. For every service type, device 6 sets the neighbor with the highest potential value as forwarding device. In this example, forwarding device for service type 'IG' is device 7. Thus, an incoming query message that searches for an Internet gateway at device 6 (or a query message composed by device 6) is forwarded to device 7.

3. If the capacity of the service instance at device 1 switches from 1 to 10, the potential field changes with the first advertisement containing the new capacity.

4. The potential value for service type 'IG' at device 2 changes to 10.83 and at device 7 to 5.33. Therefore, the forwarding device at node 6 for the service type 'IG' changes from device 7 to device 2.

MAgNETic routing forwards a query message in general toward a quantity of service instances as shown in the first part of the example, but by changing the capacity, a device can increase or scale down its influence to the routing. If a neighbor node provides a service instance, a service query searching for this service type is always forwarded to the corresponding neighbor. If the potential values of two neighbors for a service type are equal, the forwarding device is selected randomly.

2.3 Queries and Replies

To get access to a service instance, for example to send data through an Internet gateway into the Internet, a node sends a query message. This query message is routed according to the routing table entries (using the computed potential field) to the optimal service instance.

Different applications have different communication requirements, for example a sensor only needs to send data to a server while a client using an gateway to access the Internet needs a permanent bidirectional routing path to the Internet gateway. Therefore, three types for end-to-end communication between a client and a service instance exist:

- **Unconfirmed.** The communication type 'unconfirmed' is used to send data to a service instance and no reply is requested. Two subsequent query message may be delivered to different service instances of the same service type.

- **Acknowledged.** When a service instance receives a query message with communication type 'acknowledged', it sends a *reply message* back to the client. Thus, the client knows at which service instance the sent query message arrived and that it actually reached one. Again, an additional query message can be delivered to a different service instance.

- **Complete.** The client establishes a permanent routing path to a service instance and from now on always communicates with the same service instance (using query messages).

The communication types 'acknowledged' and 'complete' are answered with a reply message. In order to deliver this reply message back to the client, a new service type (called *reply service type*) is needed because there is no client network address that can be used to forward the reply message. The reply type is randomly generated, hence the client acts as the only service instance of this new generated service type. As for regular service types, the reply service type is advertised in the network and the potential field to forward reply messages is established.

Chapter 3

Protocol Specification

This chapter describes the MAgNETic routing protocol specifications in detail. This includes the MAgNETic protocol header, the specific MAgNETic protocol messages and the message handling at a node.

3.1 Preliminaries

3.1.1 Data Types

The MAgNETic protocol introduces some new data types. The following list describes the most important ones:

- UID

  Every service instance and application creates a 128 bit *unique identifier (UID)* on start up. This random generated number acts as service identifier and as a 'port number' to distinguish the different applications running at a device. The UID is with high probability unique in the whole network. The probability to get 2 equal UIDs when creating them randomly is

  $$p(UID_1 = UID_2) = \frac{1}{2^{128}} \approx 2.94 \cdot 10^{-39}$$

  In a network with 10'000 nodes, each creating 100 UIDs, the probability of having a collision (two equal UIDs) is still very low ($2.94 \cdot 10^{-33}$). If it happens nonetheless, the protocol functionality is not affected because the UIDs are not used to route packets, but it is possible to get undeterministic outcomes, for example that two different service instances are treated as the same. If an instance detects that its UID is used by another one, it should delete its UID and randomly generate a new one.

- Service Type

  A *service type* is a 128 bit number that uniquely defines the kind of service a service instance provides. Two different categories of service types exist:

  – Well-defined service types

    If the first 96 bit of a service type are equal to 0, this service is called a *well-defined* service type. The remaining 2^{32} service types are reserved for specific services. These service types for common services such as printers for example could be allocated by a specific organization.

    The number of different well-defined service types is limited to 2^{32}, but that is more than 4 billion and should satisfy the demand for service types for a long time.

  – Random generated service types

    A device can compose new service types, for example for services that were not defined before or to establish a new routing path between two devices (see section 3.5.3, created on start up). Therefore, a *random service type* is created by setting the last 32 bits to zero and randomly generates the first 96 bits. The probability to get two equal random service types is

    $$p(RST_1 = RST_2) = \frac{1}{2^{96}} \approx 1.26 \cdot 10^{-29}$$

    Hence, it is a very rare event to choose two equal random service types even in large networks (with several thousand devices). If by accident two clients choose the same random service type, that can cause trouble with routing. A packet could be forwarded to the wrong device while the intended destination device does not get the expected packet. In the worst case, the application has to create a new random service type and start communication again.

- Capacity

  Every service type is associated with a *capacity* value (16 bit integer). The capacity denotes the 'amount' of this service type a service instance offers. For example, the capacity of the service type 'Internet Gateway' could denote the bandwidth of the service instance's Internet connection.

- Potential Value

  The *potential value* (32 bit floating point) is computed with all received advertisement messages of a service type using the distance function (section 3.1.2).

3.1.2 Configuration Parameters

The MAgNETic protocol provides several *configuration parameters* to define its behavior. The values currently set allow a stable execution but need not to be the optimal selection.

- Distance Function

  To calculate the potential field, a *distance function* is used that computes the potential value given the capacity c and the distance d (number of hops) from the service instance of a received service advertisement message. The current implementation uses the function $\varphi(c,d) = \frac{c}{d}$. The potential of a service type at a node is calculated by adding up the potentials of all advertisement with this service type. The resulting function is: $\phi(c,d) = \sum_i \varphi_i(c_i,d_i) = \sum_i \frac{c_i}{d_i}$.

  It is topic of future work to evaluate well suited distance functions for different use cases, for example the function $\varphi(c,d) = \frac{c}{d^k}$ could be used, where k is the parameter to change.

- Protocol Parameters (set in file `./source/defines.h`)

  - SERVICE_AD_RATE (7 seconds). Defines the time period until a new advertisement message is sent.
  - NEIGHBOR_EXCHANGE_RATE (10 seconds). Defines the time period until a new neighbor exchange message is sent.
  - SERVICE_AD_TIMEOUT (40 seconds). An advertisement is deleted after this time if no update was received.
  - NEIGHBOR_TIMEOUT (60 seconds). If no new neighbor exchange message arrives within this time, the neighbor is deleted from the neighbor list.
  - QUERY_RESEND_TIMEOUT (5 seconds). Time to wait until an unacknowledged query message is resent.
  - MAX_QUERY_RESEND (3). Defines the maximum number of trials to send a query message if no reply is received. Afterwards, the query message is dropped.

3.1.3 Packet Overview

Figure 3.1 shows the composition of a MAgNETic routing packet. The first part is the MAC header which is processed by the network adapter. It consists of the network addresses of the sender and the receiver and a *type* value (16 bit). The type value defines the higher level protocol, currently the value (`0x08 0xff`) denotes that it is a MAgNETic packet.

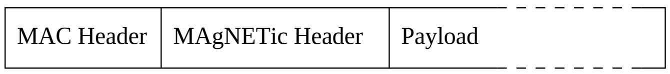

Figure 3.1: Packet Overview

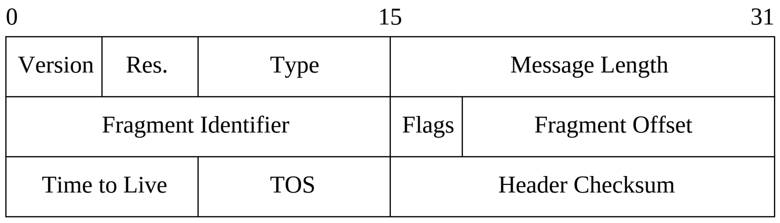

Figure 3.2: MAgNETic protocol header

Subsequent to the MAC header, the *MAgNETic header* (see section 3.2) follows. The payload, following the MAgNETic header, consists of one of the four possible MAgNETic message types (see section 3.3) and an arbitrary amount of data. The maximum length of a packet is limited by the MTU (maximum transfer unit) of the link.

3.2 MAgNETic Protocol Header

The MAgNETic protocol header is very similar to the IP header [5] with the major difference that no network addresses are included. It contains the following fields (figure 3.2):

- *Version (4 bit).* Version of the protocol, currently 1.
- *Reserved (4 bit).* Reserved for further use.
- *Packet Type (8 bit).* MAgNETic type of the packet, defines the subsequent payload. Currently, for different packet types exist (see section 3.3):
  - Type 1: `MG_SERVICE_AD`: Advertisement Packet
  - Type 2: `MG_NEIGHBOR_EXCHANGE`: Neighbor Exchange Packet
  - Type 3: `MG_SERVICE_QUERY`: Query Packet
  - Type 4: `MG_SERVICE_REPLY`: Reply Packet
- *Message Length (16 bit).* Length of the payload (MAgNETic packet and optional data, without the MAgNETic and the MAC header).

- *Fragment Identifier (16 bit).* Random number to identify the fragments of the same message. If no fragmentation is used, the fragment identifier is set to 0.

- *Fragmentation Flags (3 bit).* Three fragmentation flags can be set (according to the IP protocol):

  - `R: Reserved`, must be 0.
  - `DF: Do not Fragment`, set to 1 if the packet can not be fragmented. Currently unused.
  - `MF: More Fragments`, set to 1 if at least one more fragment follows, set to 0 if this is the last fragment of the message.

  Currently only the third flag (`MF`) is considered.

- *Fragmentation Offset (13 bit).* Offset of this fragment in the entire packet (in blocks of 8 bytes).

- *Time To Live (8 bit).* To avoid endless loops in the network, a time to live field is included. The composer of a packet sets this field to an initial value (currently 32) and every time a packet is forwarded, the TTL is decremented by 1. A packet will not be forwarded if its TTL equal to 0 after decrementing.

- *Type of Service (8 bit).* Currently unused (set to 0).

- *Header Checksum (16 bit).* The header checksum serves to check if the MAgNETic header was correctly transmitted. The MAgNETic protocol uses exactly the same function to compute the checksum as the IP protocol [5].

The fragmentation is used when a relaying node provides more than one network interface with different MTUs (Maximum Transfer Units). It is possible that a packet received from one network interface needs to be forwarded out of the other interface with a smaller MTU. If the packet size exceeds this smaller MTU, it is necessary to split the packet into several fragments. The goal of an implementation is to use fragmentation as less as possible. If all devices limit the packet size to 1'500 bytes (MTU of an Ethernet), fragmentation should not be necessary at all.

3.3 Message Types

To keep routing tables and service information of all devices in the network up to date, each device periodically sends two types of messages: *advertisement messages* and *neighbor exchange messages.* These messages are sent

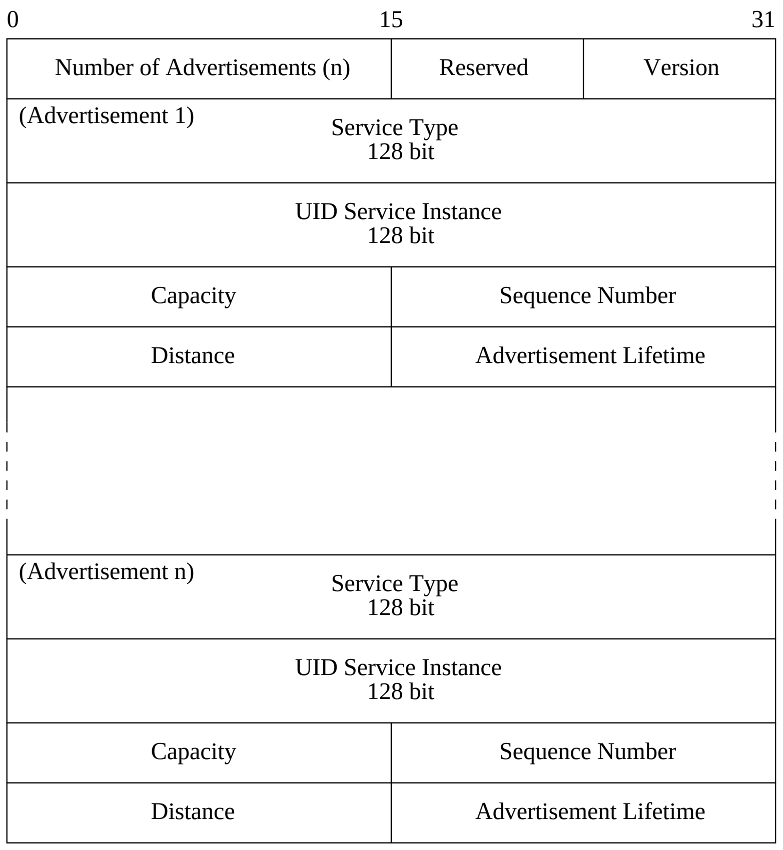

Figure 3.3: Advertisement Message

broadcast to all devices in range. An advertisement message contains information about the service types provided by the transmitting device and is forwarded in the entire network, the neighbor exchange message contains the actual routing table (all potential values) of a device and is sent to all devices in range.

Additionally to these periodic update messages, two other message types are provided: *query messages* and *reply messages*. A client sends a query message to get access to a service instance of the desired service type. When a device receives a query message for a service type that it provides, a reply message is sent back to the client.

3.3.1 Advertisement Message

In order to avoid too much control overhead, several service descriptions can be collected in an *advertisement message*. Therefore, an additional header is prepended to the single service advertisements. This header contains the following fields (figure 3.3):

- *Number of Advertisements (16 bit).* Number of single advertisements (n) included in this message.
- *Reserved (8 bit).* Reserved for further use.
- *Version (8 bit).* Version of the service advertisement message (current version: 1).

The n advertisements are appended to the message header, each consisting of the following fields:

- *Service Type (128 bit).* Service type of the provided service.
- *UID Initiator (128 bit.)* UID of the providing service instance.
- *Capacity (16 bit).* Capacity of this service type provided by the service instance.
- *Sequence Number (16 bit).* Sequence number of the advertisement. Every time a device sends a advertisement message, it increments the sequence number of this advertisement. The sequence number is used to detect duplicate advertisements, for example if the same advertisement arrives twice over different paths.
- *Distance (16 bit).* Distance (number of hops) to the service instance.
- *Advertisement Lifetime (16 bit).* Value to limit the distribution range of an advertisement. The service instance sets the advertisement lifetime (currently to 8) and every node decrements the lifetime before forwarding the advertisement. An advertisement with lifetime equal to 0 is used to compute the potential value but not forwarded any more.

For further improvements, collecting several received advertisements and forward them in a single advertisement message reduces the overhead traffic in the network by a remarkable amount. This technique, described in [3] is topic of future work.

3.3.2 Neighbor Exchange Message

Neighbors periodically exchange their potential values for all service types among each other. With the potential values of all neighbors in range, a device can build up its routing table. For every service type, the neighbor with the highest potential value is set as forwarding device to route query message.

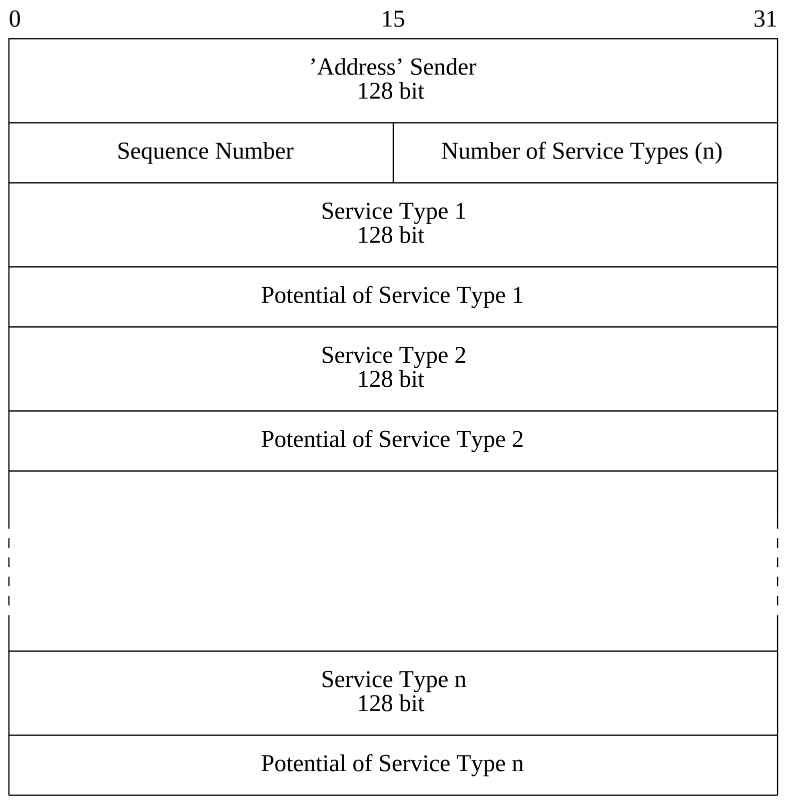
0
15
31
'Address' Sender
128 bit
Sequence Number
Number of Service Types (n)
Service Type 1
128 bit
Potential of Service Type 1
Service Type 2
128 bit
Potential of Service Type 2
Service Type n
128 bit
Potential of Service Type n

Figure 3.4: Neighbor Exchange Message

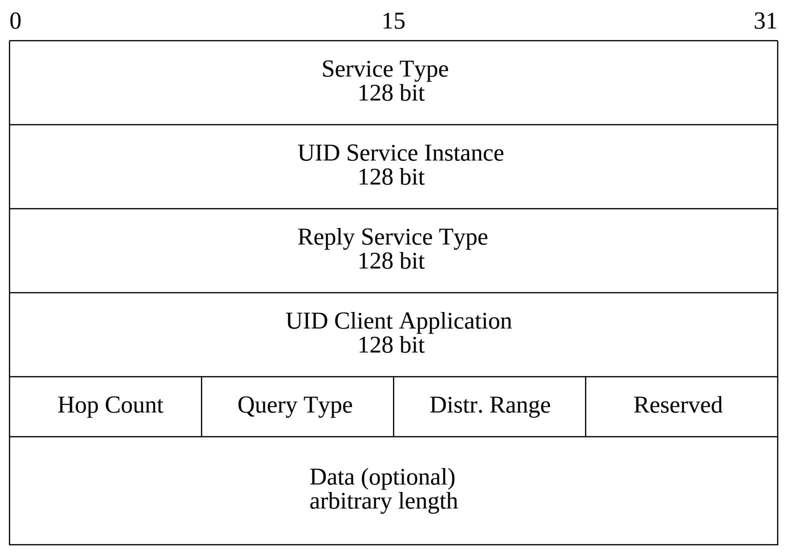

Figure 3.5: Query Message

The neighbor exchange message starts with a short header to denote the number of service types and corresponding potential values included in this packet (figure 3.4):

- *'Address' Sender (128 bit).* Reply service type of the distributing device (see section 3.4.4).
- *Sequence Number (128 bit).* The sequence number is incremented with every change of the routing table. If a device remains in a stable state with no change of potentials, the sequence number remains the same. The receiving device can check the sequence number to determine if any potentials in this neighbor's routing table have changed.
- *Number of Services (16 bit).* Number of service types in the message.

All service types currently existing in the sender's routing table are appended to this header, each beginning with the *Service Type (128 bit)* followed by the *Potential Value (32 bit).*

For further improvements, the link quality to each neighbor could be saved and considered before forwarding a query message.

3.3.3 Query Message

A client sends a *query message* to access a service instance. The query message is routed (according to the potential field of the service type) through the network to a device that provides the specified service type. A query message contains the following fields (figure 3.5):

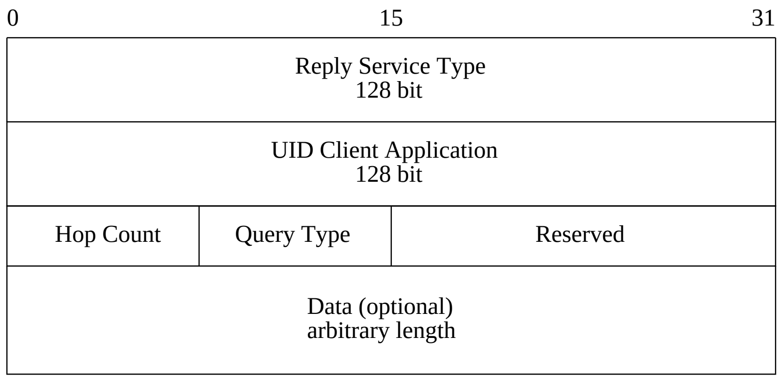

Figure 3.6: Reply Message

- *Service Type (128 bit).* Service type to search for.
- *UID Service Instance (128 bit).* UID of the service instance. Set to 0 if not known when sending the query.
- *Reply Service Type (128 bit).* If the client asks for a reply (query type 2 or 3), a reply service type is used to route back the reply message (for more details, see section 3.4.4).
- *UID Client Application (128 bit).* UID of the application that sends the query. Used to deliver a reply to the application.
- *Hop Count (8 bit).* Distance (number of hops) to the client.
- *Query Type (8 bit).* The query type defines the behavior of sender, intermediate nodes and receiver of the query message (see section 3.4).
- *Distribution Range (8 bit).* Limits the range a reply service type is advertised in the network. If the distribution range is set to 0, a reply message is forced to take the same route as the query message. For more details, see section 3.4.4.
- *Reserved (8 bit).* Reserved for further use.
- *Data (arbitrary length).* A service query message contains data of arbitrary length. To determine the length of the appended data, the field *message length* in the MAgNETic header (see section 3.2) is used (data length = message length - size of query message without data).

3.3.4 Reply Message

A service instance receives a query message from a client and decides what action to take. One possibility among others (see sections 3.4 and 3.5.3) is to send a reply message back to the client. The reply message contains the following fields (figure 3.6):

- *Reply Service Type (128 bit).* The reply service type is used to forward the reply packet back to the client.
- *UID Client Application (128 bit).* UID of the client application to deliver the reply.
- *Hop Count (8 bit).* Distance (number of hops) from the service instance.
- *Query Type (8 bit).* The query type defines the behavior of sender, intermediate nodes and receiver of the reply message (see section 3.4).
- *Reserved (16 bit).* Reserved for further use.
- *Data (arbitrary length).* A reply message contains data of arbitrary length.

3.4 Communication Types

Considering the end to end communication between a client and a service instance, different kind of applications need varying action of the participating nodes. To satisfy the different requirements, the MAgNETic protocol provides three *communication types* (also called *query types*):

- Type 1: Unconfirmed
- Type 2: Acknowledged
- Type 3: Complete

The communication type is set in the field `query type` in the *query message* (section 3.3.3) or *reply message* (section 3.3.4). Depending on the communication type, the intermediate devices execute different tasks while forwarding a query or reply message (detailed description in section 3.5.3).

3.4.1 Communication Type: Unconfirmed

The communication type *unconfirmed* can be used for example to send sensor data to a service instance. As shown in Figure 3.7, a client sends a query to a service instance using the advertised service type. The receiving service instance delivers the query to the corresponding application and no further action is taken. The client does not know the identity of the receiving service instance and has even no guarantee that the query message actually reached a service instance. Also, subsequent messages can be delivered to different service instances if the potential field changed in the meantime.

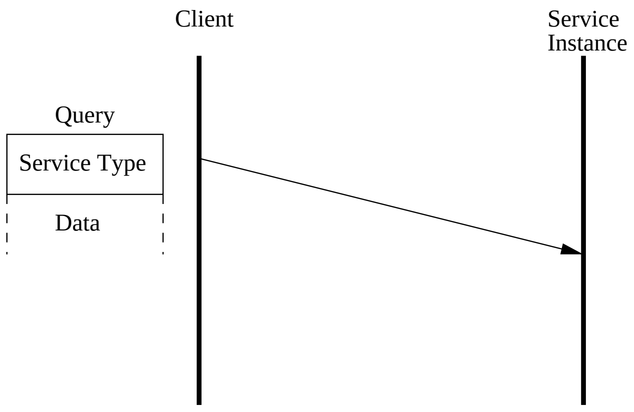

Figure 3.7: Communication Type: Unconfirmed

3.4.2 Communication Type: Acknowledged

If the client needs a reply to the query message, it uses the communication type *acknowledged*. The receiving service instance answers with a reply message that among other fields contains its UID (see figure 3.8). Therewith, the service instance is uniquely identified.

The client generates a new service type *reply type* as a random service type (section 3.1.1). A new potential field for the service type *reply type* is established. The service instance sends the reply message to a provider of the service type *reply type*. Because the client is the only service instance for that reply type, the reply message will be routed back to the client. Again, there is no guarantee that two subsequent query messages arrive at the same service instance.

3.4.3 Communication Type: Complete

Communication type *complete* serves to establish a permanent routing state between two devices (see figure 3.9). Unlike with type acknowledged, two new service types (*reply type 1*, *reply type 2*) and therefore two new potential fields are used. After the set-up phase with one query message each, the client has established a stable routing state to a service instance. Henceforth, the two devices communicate with each other using the two new reply service types.

3.4.4 Reply Routing

The client (query types 2 and 3) and the service instance (query type 3) use a random service type *reply type* that is distributed in the network with an advertisement message. Therewith, the potential field for the reply type

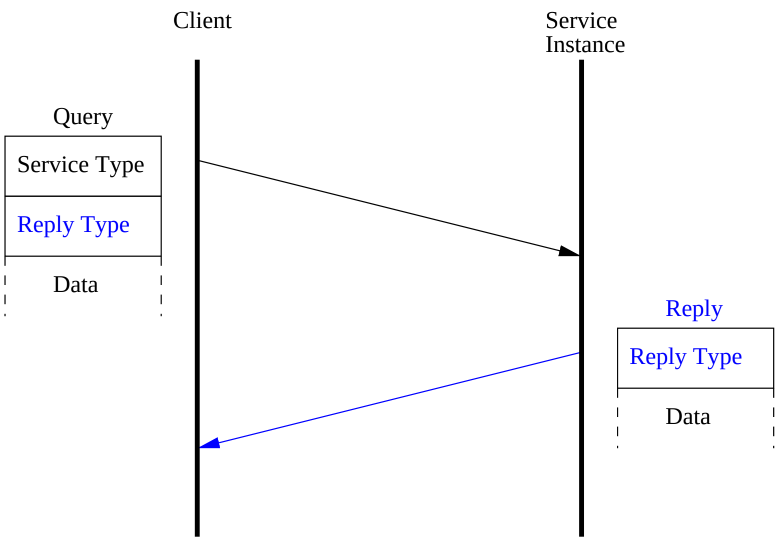

Figure 3.8: Communication Type: Acknowledged

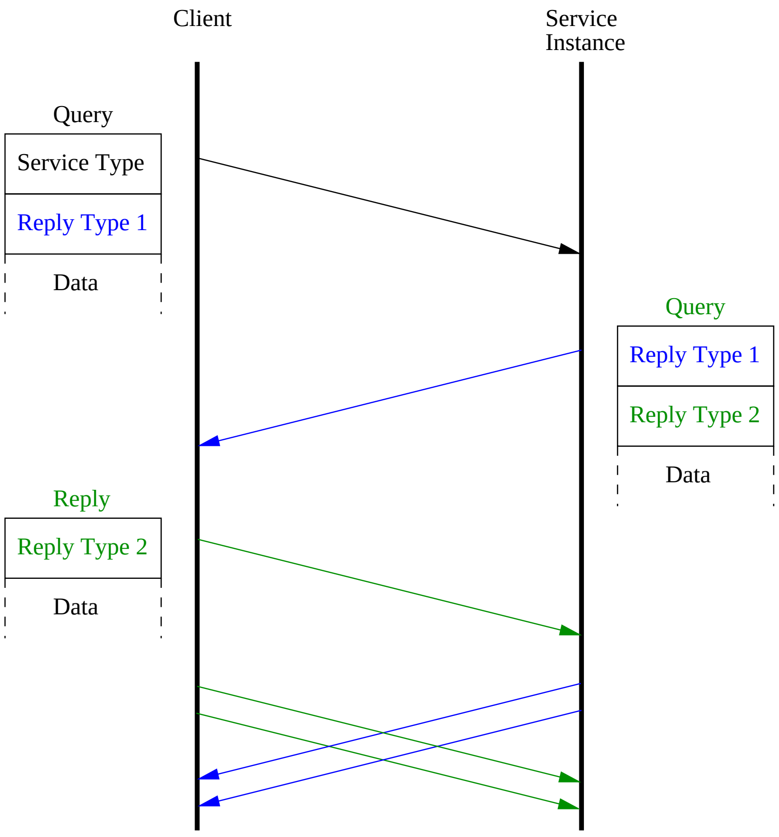

Figure 3.9: Communication Type: Complete

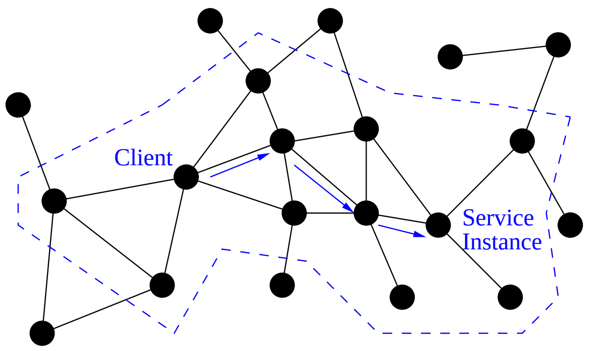

Figure 3.10: Distribution Range of a Reply Service Type

is built. Because establishing the potential field possibly takes too much time to route back the reply message (if the reply message arrives before the routing table entry is updated), a device that forwards a query message composes a routing table entry for the reply type: it sets the node that the query message was received from as forwarding device for the reply type. If the reply message arrives before the potential field is established, it takes the same path as the query message back to the client.

To limit the range a reply type is distributed in the network, the field *distribution range* in the query message (see section 3.3.3) is used. Figure 3.10 shows the distribution of a reply service type with distribution range of 1:

- A client sends a query message to a service instance.
- Client, service instance and all relaying nodes send an advertisement for the service type *reply type* with an *advertisement lifetime* of 1, which specifies the distribution range.
- Therefore, the reply type is distributed in the range of 1 hop from the path of the query message (the region inside the dashed line).

Note, that the current implementation does not allow to use the distribution range. The reply types are advertised by the initiator (client or service instance) and sent broadcast with the default advertisement lifetime.

3.5 Message Handling

Advertisement and neighbor exchange messages are used to build up the routing tables, while query and reply messages are used to access a service instances and to send data. This section describes the handling of incoming MAgNETic messages at a node.

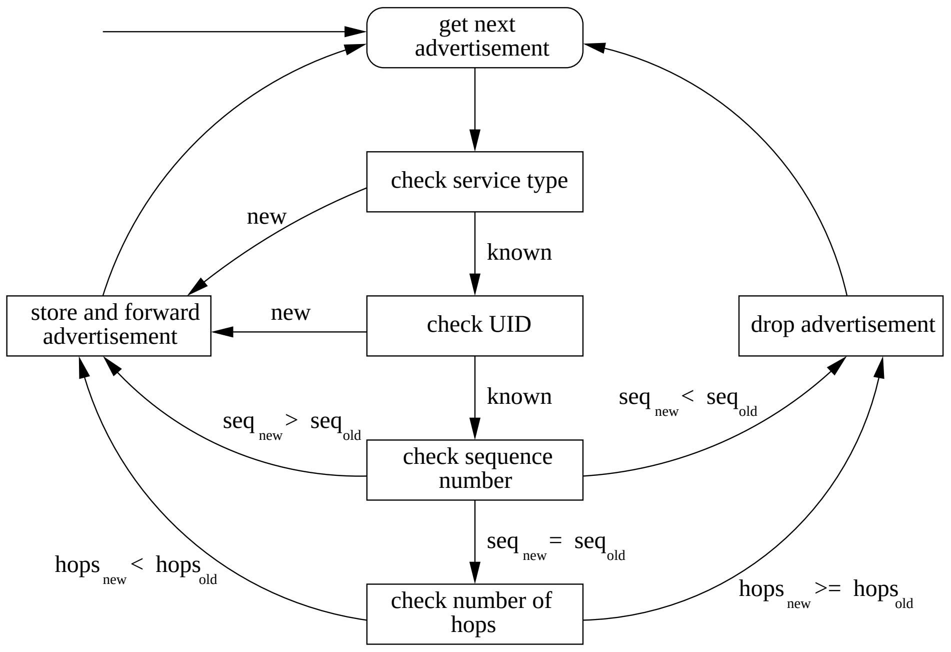

Figure 3.11: Advertisement Handling

3.5.1 Service Advertisement Handling

Because all advertisement messages are sent broadcast in the network, it is possible that a device receives several times the same advertisement. In this case, the duplicate message is dropped. If the advertisement is new, the device has to update its routing table and store the advertisement. All new advertisements need to be forwarded broadcast to all devices in range with one exception: if the *advertisement lifetime* (see section 3.3.1) is equal to 0, the advertisement timed out and is dropped. The advertisement lifetime is set to an initial value at the service instance (actually 32) and decremented by every device before forwarding. Therewith, it is possible to limit the distribution range of an advertisement in the network.

The handling of an incoming advertisement is shown in figure 3.11. If the *service type* of the received advertisement does not exists in the device's routing table, the advertisement is stored and forwarded. If the service type is already known, the *UID* of the service instance is checked. A new UID denotes that the existing entry with this service type in the routing table results from a different service provider of the same service type. Therefore, the advertisement is stored and forwarded.

If both service type and UID are known, it is about another copy of an existing advertisement. The device then compares the received advertisement to the stored one. If the *sequence number* is higher, the advertisement

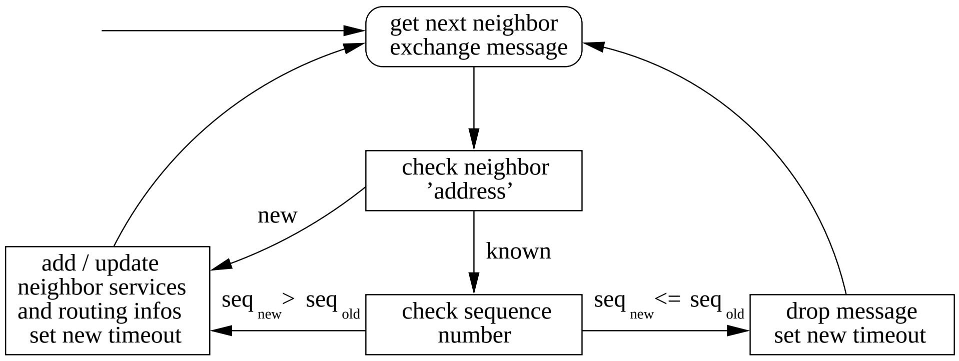
get next neighbor exchange message
check neighbor 'address'
new
known
add / update neighbor services and routing infos set new timeout
$seq_{new} > seq_{old}$
check sequence number
$seq_{new} <= seq_{old}$
drop message set new timeout

Figure 3.12: Neighbor Message Handling

is newer and therefore replaces the old one; the new advertisement is stored and forwarded. If the sequence number is smaller, the received advertisement is dropped. If the sequence numbers are equal, the same advertisement arrived twice over different paths, hence the *number of hops* is considered. If the new advertisement took a shorter path to this device (less hops), it replaces the old advertisement. Otherwise, the received advertisement message is dropped.

3.5.2 Neighbor Exchange Handling

Every device periodically sends neighbor exchange messages that contain all service types and the corresponding potential values existing in the routing table. A device stores all neighbors and their potential values for all service types in a neighbor list. If a neighbor disappears, the corresponding data is deleted after a timeout time. The neighbor exchange message contains a sequence number that shows the status of the information: every time an entry in the local routing table changes, the sequence number is incremented.

The handling of an incoming neighbor exchange message is shown in figure 3.12. First, the neighbor's *address* (the reply service type) is checked. If this neighbor is already known, the *sequence number* in the message is checked and the message is dropped if the new sequence number does not exceeds the stored one.

If either the neighbor is new or the sequence number denotes a change in the neighbor's routing table, the receiving device has to execute the following tasks:

- Add the neighbor to the neighbor list if it is a new neighbor.
- Update the neighbor's potential values for all service types.
- Update the local routing table if necessary: for every service type,

	Type 1 unconfirmed	**Type 2** acknowledged	**Type 3** complete
Sending a Query	send query	send query save in list send ad (reply type)	send query save in list send ad (reply type)
Forwarding a Query	forward query	forward query set routing infos send ad (reply type)	forward query set routing infos send ad (reply type)
Receiving a Query	→ application	→ application compose reply	→ application compose query save in list
Sending a Reply		send reply	(send query) complete list send ad (new reply type)
Forwarding a Reply		forward reply	(forward query) set routing infos send ad (new reply type)
Receiving a Reply		→ application remove from list	→ application complete list

Table 3.1: Query and Reply Handling

the neighbor with the highest potential value is set as forwarding device. The device checks for all service types in the neighbor exchange message if the forwarding device needs to be changed.

- Set a new timeout value for this neighbor. If no new neighbor exchange message is received within this time, the neighbor is deleted from the neighbor list.

3.5.3 Query and Reply Handling

A client sends a query message to get access to a service instance. The query type (see section 3.4) determines the behavior of all involved devices (client, service instance and relaying nodes). Table 3.1 summarizes the main tasks for a device when sending, forwarding or receiving a query or a reply message. The following paragraphs describe the tasks in detail:

- Sending a Query

  The MAgNETic protocol gets an instruction from an application to send a *query message* (with data appended) to a service instance of a specific service type. It sends the message to the neighbor with the highest potential value for the desired service type (according to the routing table). If no routing table entry exists for this service type, the application is notified and the query message is dropped.

Query types 2 and 3: The query is stored in the *query list* to control if a reply message arrives. If no reply arrives within a specific time, the query message is sent again. If still no reply arrives after a specific resend trials, the failure is reported to the application and the query message is dropped. A service advertisement for the service type *reply type* is sent to establish a potential field to route back the reply message.

- Forwarding a Query

  After receiving a query message, a device checks if the desired service type is provided by itself. If not, the query message is forwarded to the neighbor with the highest potential value for this service type. If no routing table entry for the desired service type exists, the query message is dropped.

  Query types 2 and 3: The relaying node sets the device from that the query message was received as forwarding device for the service type *reply type.* With this routing table entry, it is guaranteed that the reply message is routed back to the client even if it arrives before the potential field for the service type *reply type* is established. To limit the distribution range of the service type *reply type* (see section 3.4.4), all relaying nodes send an advertisement for the service type *reply type.*

- Receiving a Query

  If the device provides the desired service type, the query message and the appended data is delivered to the corresponding application.

  Query type 2: Every query message of type 2 has to be acknowledged with a *reply message.* Thus, the device composes a reply message and sends it back to the client.

  Query type 3: The device sends a new query message back to client, that contains a new service type *new reply type.* Both the *reply type* and the *new reply type* are saved in the query list and used further on for the communication between this service instance and the client.

- Sending a Reply

  Query type 2: The service instance sends a reply message back to the client

Query type 3: A new query is sent as an answer to a query message with query type 3. The service instance sends an advertisement for the new service type *new reply type*.

- Forwarding a Reply

  Query type 2: The reply message is forwarded using the *reply type* entry in the routing table.

  Query type 3: The new query message is forwarded using the *reply type* entry in the routing table. To limit the distribution range of the service type *new reply type* (see section 3.4.4), all relaying nodes send an advertisement for the service type *new reply type*.

- Receiving a Reply

  Query type 2: The reply message and the appended data is delivered to the corresponding application and the entry in the query list is deleted (a connection using query type 2 is complete after receiving a reply message).

  Query type 3: The query message and the appended data is delivered to the corresponding application. The query list entry is completed with the *new reply type*. Further on, the client and the service instance communicate using the *reply type* and the *new reply type*.

Note, that the current implementation does not allow to use the distribution range of an advertisement (see section 3.4.4). The reply service types are advertised only by the initiator (client or service instance) with the default advertisement lifetime. Hence, when forwarding a query or a reply message (type 2 or 3), the relaying nodes do not send an advertisement.

Chapter 4

Implementation

This chapter describes the basic considerations of the MAgNETic routing protocol implementation. The implementation follows the following design principles:

- **Platform independent**. Written in ANSI C.
- **Non blocking**. Execution in several threads.
- **Direct link-layer access**. Network access with raw sockets without going through the TCP/IP stack.

A guide for installing and starting the MAgNETic protocol and the developed tools is given in the appendix.

4.1 Data Structures

The most important data structures are the *Routing Table* (contains all routing infos as well as all received advertisements) and the *Neighbor List* (with all neighbors and their potential values for all service types).

4.1.1 Routing Table

The main data structure of the MAgNETic routing protocol is the routing table. To guarantee a fast access, it is organized as a *hash table*, currently of size 64. The hash value is computed with the service type. Collisions in the hash table (service types that result in the same hash value) are stored in a list with the following entries:

```
struct services_list
{
   uid service_type;                 // service type
   float potential;                  // potential value
   int own_service;                 // set to 1 if this node
                                    //   provides service type
   unsigned char next_hop_mac[6];   // forwarding MAC address
   float next_hop_capacity;         // potential value at
                                    //   forwarding neighbour
   struct ad_list * advertisements; // list with all ads
                                    //   of this service type
   struct services_list * next;     // pointer to next entry
}
```

The MAgNETic protocol stores all received advertisements in the routing table. If a new advertisement arrives or an existing one times out, the potential value of the corresponding service type is recalculated. To complete the routing table entry, the MAgNETic protocol chooses the forwarding 'next hop MAC' address and the corresponding 'next hop capacity' value from the neighbor list.

4.1.2 Neighbor List

Using the incoming neighbor exchange messages, every device stores all neighbors in the neighbor list with the following entries:

```
struct nb_list
{
   uid neighbor_uid;                // UID of this neighbor
   unsigned char mac[6];            // MAC of this neighbor
   short sequence_number;           // sequence number
   struct timeval timeout;          // timeout value
   struct nb_services * nb_services // list with all service
                                    //   types and potentials
                                    //   in the neighbor's
                                    //   routing table
   struct nb_list * next;           // pointer to next entry
}
```

The sequence number serves to check if any potential value in the neighbor's routing table changed (every device increments its neighbor exchange sequence number with every change of its routing table). The neighbor entry contains a list with all service types and the corresponding potential values in the neighbor's routing table. This information is used to set the neighbor with the highest potential value as forwarding device in the routing table (for every service type). If no new neighbor exchange message is received within a specific time, the neighbor is deleted from the list.

4.1.3 Additional Data Structures

Some more data structures are required, the most important are:

- **Timer List.** Contains all pending timers, each with a *timeout time* and a *handling value* that defines the required handling routines.
- **Query List.** The query list contains all open routing paths from this device to others. When a device sends a query to a service instance or receives one as a service instance from a client, it composes a query list entry to control the communication between the two devices. As soon as the communication between these two devices is finished, the query list entry is removed.
- **Fragments List.** Fragmented packets are saved in the fragments list until all fragments of a message arrived. Only if a message is complete, it is delivered to the corresponding thread. This list is cleaned from incomplete messages periodically.
- **Own Services List.** This list contains all service types provided by this device and the corresponding capacity.

4.2 Networking

On start up, the user defines the network interface to use. Normally, this is a wireless LAN network interface, but also a LAN interface can be used, for example for testing and debugging. To get access to a network interface, a socket [11] to the network interface is opened. All MAgNETic packets are of the following structure (see section 3.1.3):

- MAC header
- MAgNETic header
- Payload (arbitrary length)

Instead of using the TCP/IP stack, the MAgNETic protocol uses an own MAgNETic header. A *raw socket* is opened to communicate with the network interface. Therewith, it is possible to access the data link layer directly and skip the IP and TCP layers (see figure 4.1).

The network interface delivers all received packets to the raw socket. Thus, the MAgNETic protocol has to check itself if the received packet is a MAgNETic packet or drop it otherwise. A specific value (currently `0x08 0xff`) in the `type` field of the MAC header indicates that the received packet actually is a MAgNETic packet.

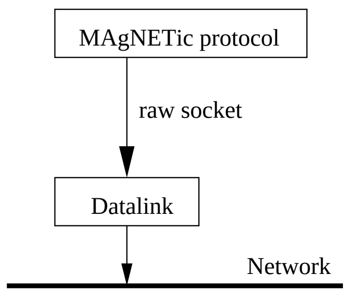

Figure 4.1: Raw Sockets

4.3 Threads

The MAgNETic protocol starts five threads at the beginning, each responsible for a specific type of tasks:

- **Receive thread:** Receives the packets from the network interface and either forwards them to another device according to the routing table or delivers them to the corresponding thread for further handling.
- **Magnetic thread:** Gets all advertisement and neighbor exchange messages from the receive thread and uses them to build up the routing table.
- **Timer thread:** Responsible for all periodic action: sends periodic advertisement and neighbor exchange messages and continuously removes old data.
- **Query thread:** Handles all query and reply messages and is also responsible for the communication between the MAgNETic protocol and the appropriate applications (service instances and service consumers).
- **Interaction thread:** Waits for a user input and executes the desired action, used to access a service instance (send a query) and for debugging tasks.

The main advantage of using different threads is to execute several tasks in parallel (or pseudo-parallel if there is in fact just one processor available). Therefore, a task will not block another. For example, the packet forwarding should be executed as fast as possible and not be blocked by a less important (and not time critical) task like periodic cleaning up. Using several threads, the forwarding of a packet needs not to wait until the cleaning up task has finished and can be executed in parallel. In the following, the threads are illustrated in more detail and the communication between them is briefly outlined.

4.3.1 Receive Thread

Figure 4.2 shows the routines of the receive thread. It first initializes the network interface and then waits in an endless loop for incoming packets. If a packet arrives, the thread tests if it actually is a MAgNETic packet and checks it for integrity. According to the routing table, the packet is either forwarded to another device or it is intended for this node. Then, the fragmentation reassembly is carried out if necessary. The packet identifier defines the further handling: advertisement and neighbor exchange messages are delivered to the *magnetic thread*, query and reply messages to the *query thread*.

4.3.2 Magnetic Thread

The magnetic thread is responsible for the maintenance of the routing table and the neighbor list. Figure 4.3 shows the cycle of the magnetic thread: it gets all advertisement and neighbor exchange messages from the *receive thread* and handles them according to the description in section 3.5.1 (advertisement message) and section 3.5.2 (neighbor exchange message) respectively. Then the magnetic thread waits for the next message.

4.3.3 Timer Thread

On start up, the timer thread sends the first advertisement message (if the device provides at least one service type) and sets all needed timer: an advertisement send timer to resend the advertisement message, a neighbor exchange send timer to distribute the neighbor exchange message and all actualization timer to periodically check data for timeout. Then the thread starts waiting for the next timeout and handles the required tasks (see figure 4.4). For all periodic action, a new timer with the same handling tasks is set.

4.3.4 Query Thread

The query thread (see figure 4.5) gets all query and reply messages from the *receive thread*. If a query message is received, the appended data is stored in a file (at the moment, there is no application attached to a provided service type). If desired (see section 3.5.3), a reply message is sent back to the client. If a reply message is received, the appended data is delivered to the application that sent the corresponding query message.

4.3.5 Interaction Thread

As shown in figure 4.6, the interaction thread handles all user inputs. Using a pipe, an application asks for an action. Possible actions at the moment are

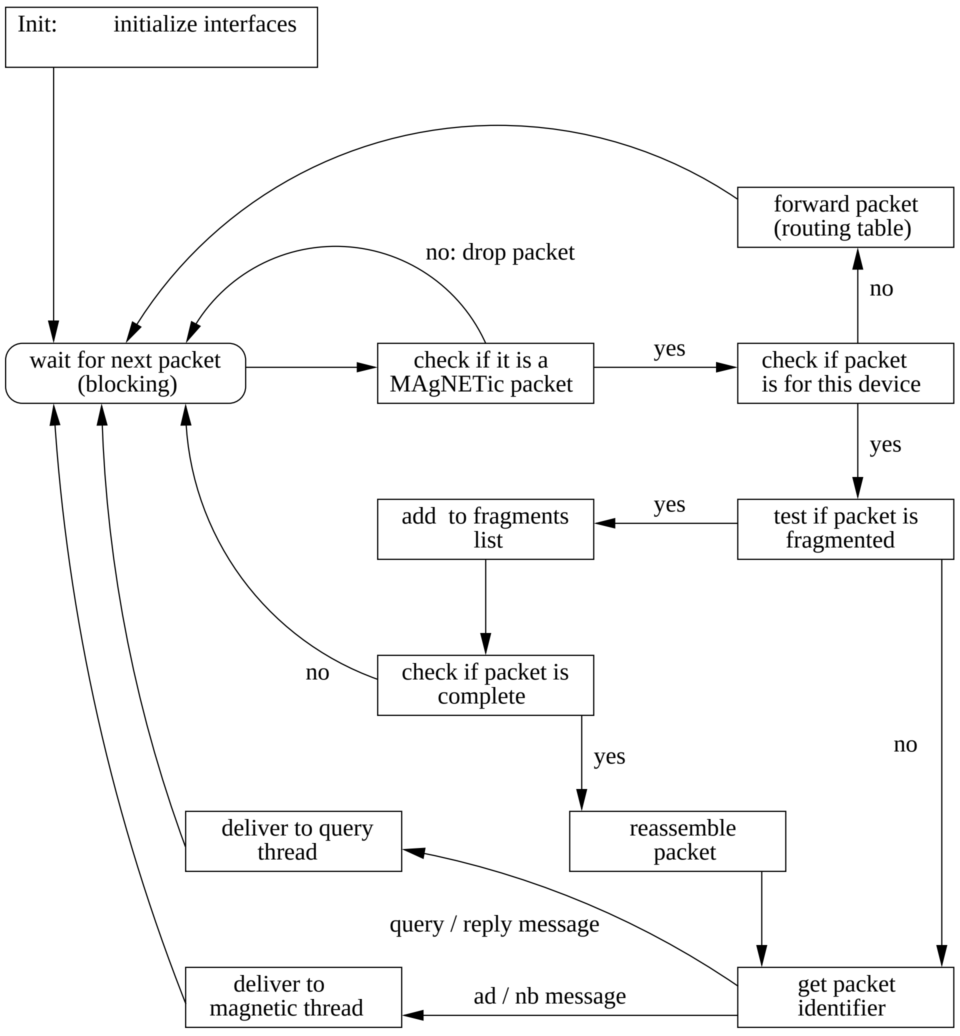
Init: initialize interfaces
forward packet (routing table)
no: drop packet
no
wait for next packet (blocking)
check if it is a MAgNETic packet
yes
check if packet is for this device
yes
add to fragments list
yes
test if packet is fragmented
no
check if packet is complete
yes
no
deliver to query thread
reassemble packet
query / reply message
deliver to magnetic thread
ad / nb message
get packet identifier

Figure 4.2: Receive Thread

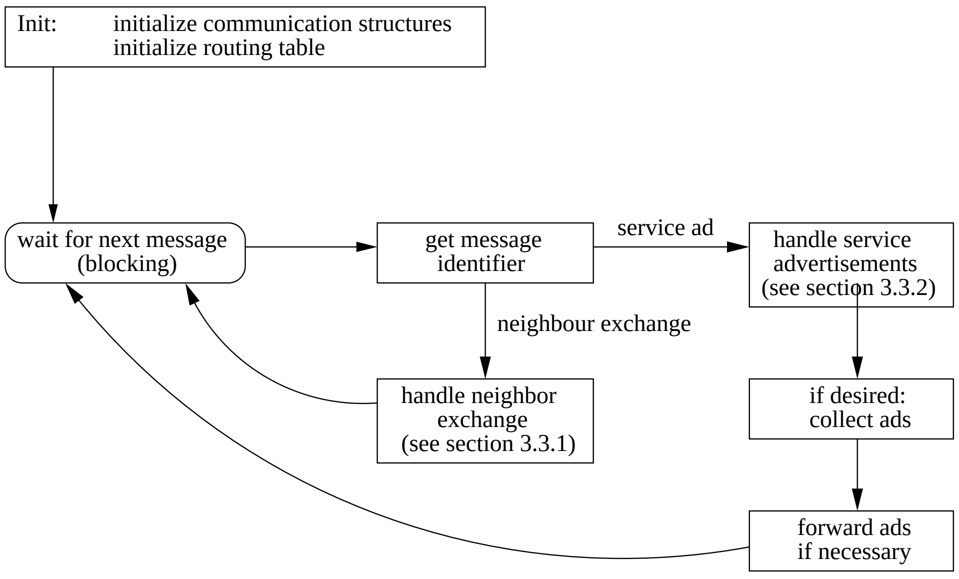
Init: initialize communication structures
initialize routing table
wait for next message (blocking)
get message identifier
service ad
handle service advertisements (see section 3.3.2)
neighbour exchange
handle neighbor exchange (see section 3.3.1)
if desired: collect ads
forward ads if necessary

Figure 4.3: Magnetic Thread

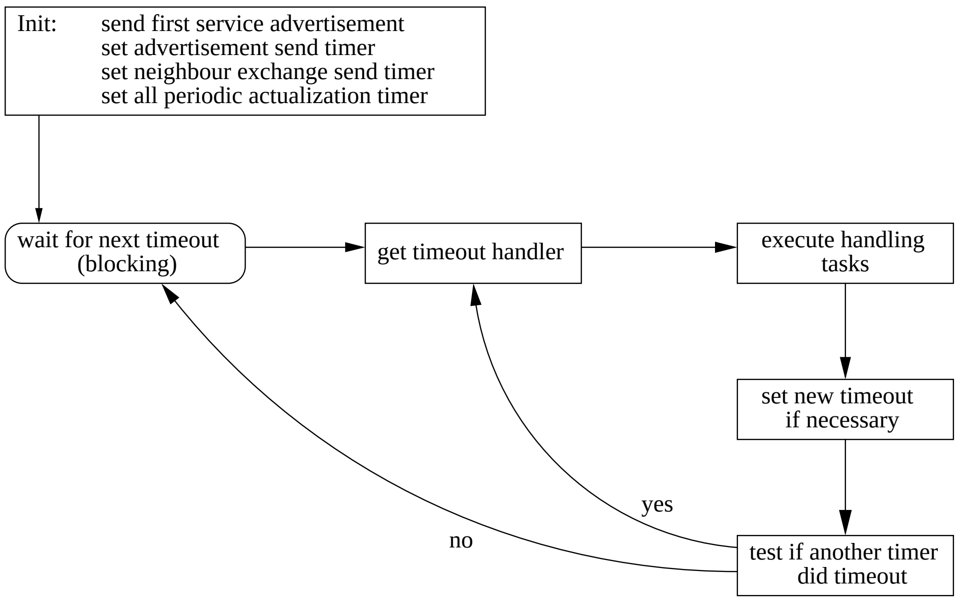
Init: send first service advertisement
set advertisement send timer
set neighbour exchange send timer
set all periodic actualization timer
wait for next timeout (blocking)
get timeout handler
execute handling tasks
set new timeout if necessary
yes
no
test if another timer did timeout

Figure 4.4: Timer Thread

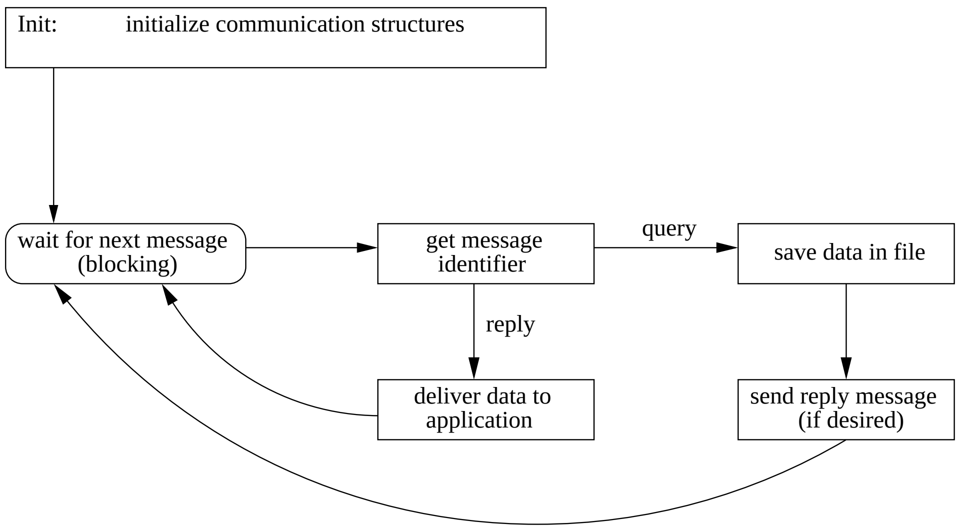
Init: initialize communication structures
wait for next message (blocking)
get message identifier
query
save data in file
reply
deliver data to application
send reply message (if desired)

Figure 4.5: Query Thread

(see appendix B): send a query message, write out intern data (for example the routing table), change the provided services or use MAC address filtering. The interaction thread processes the desired task and waits for the next user input.

4.3.6 Communication between Threads

The different threads execute independent tasks, but sometimes they need to communicate among themselves. Figure 4.7 shows the necessary thread interactions.

- The receive thread gets the packets from the network interface and delivers them to the corresponding thread (magnetic or query thread). Thus, two *unnamed pipes* are created with the receive thread on the write end and the magnetic or query thread on the read end. The magnetic and query threads are blocked until the receive thread delivers a packet to them by writing the memory address of the packet into the pipe. Then, the magnetic or the query thread handles the delivered packet while the receive thread waits for the next packet from the network interface.

- An indirect communication between the threads takes place using the routing table. The magnetic thread is responsible for composing the routing table entries while the receive, query and interaction threads use it to get the forwarding MAC address to send a packet.

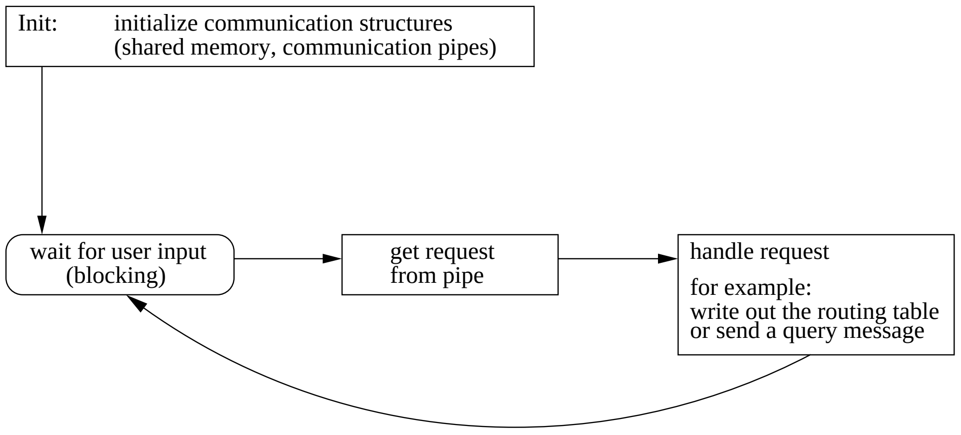
Init: initialize communication structures
(shared memory, communication pipes)
wait for user input
(blocking)
get request
from pipe
handle request
for example:
write out the routing table
or send a query message

Figure 4.6: Interaction Thread

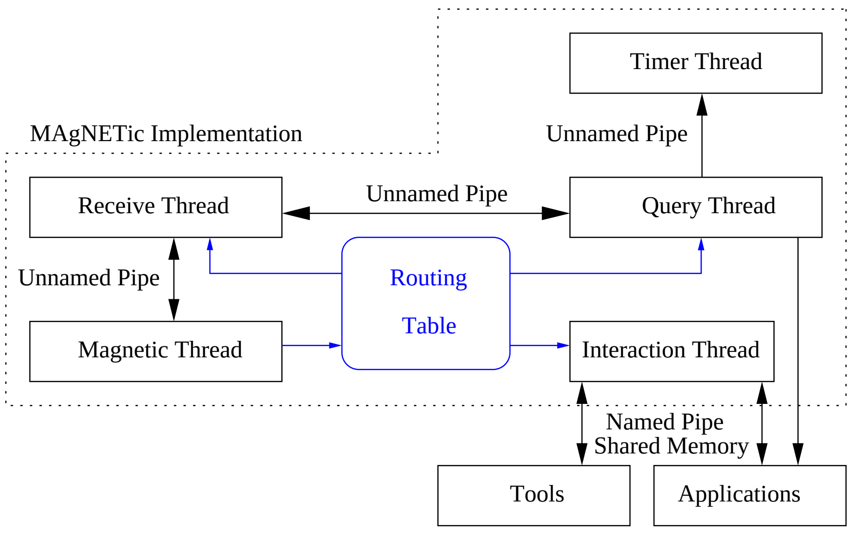

Figure 4.7: Communication between Threads

- The timer thread needs no communication with other threads with one exception: the query thread can set an additional timer to resend a query message if no answer was received within a specific time interval. To set the additional timer, an *unnamed pipe* is used.

- The communication between the interaction thread and a tool or application (see appendix B) is implemented with a *shared memory* area to deliver data and a *named pipe* to inform the magnetic protocol that there is some work to do. The interaction thread is blocked until an application writes an instruction to the pipe.

- The query thread sends a signal to the corresponding application if a reply arrived. The received data is delivered using the shared memory area. If a query message arrives for a service instance, the data is written to a file and a reply is sent automatically if desired, because at the moment, there are no application running that provide the service types.

Chapter 5

Tests

The performed tests serve to validate the implemented code and to prove that the MAgNETic routing model works in a real environment.

5.1 Environment

The MAgNETic protocol was implemented in C under Linux. It was tested with the following hardware and software components:

- **Kernel.** The basic system was Debian Linux. The implementation was tested on the two kernels 2.4.24 and 2.6.10.
- **Compiler.** As compiler, the GNU project C compiler (gcc) versions 2.95.4 and 3.3.5 were used.
- **Wireless Interfaces.** The implementation was tested with a wireless LAN adapter bases on an *Intersil Prism 2.5* chipset (Intersil Corporation Prism 2.5 Wavelan chipset (rev 01)). This interface works with the orinoco_pci driver (*orinoco_pci.o* kernel module). The wireless LAN adapter is set into the ad hoc mode and all devices of the same MANET are given the same network cell number (ESSID). No problems occurred while running the code using an interface adapter with a Prism 2.5 chipset.

  Tests with another wireless LAN adapter that uses an *Atheros* chipset (Atheros Communications, Inc. AR5212 802.11 abg NIC (rev 01)) did not work properly. This interface was set up with the *ath_pci.ko* kernel module. As soon as a packet with a unicast destination address is sent to another device, the interface adapter starts to produce continuous hardware interrupts, and after a few seconds the computer crashes and needs to be rebooted.

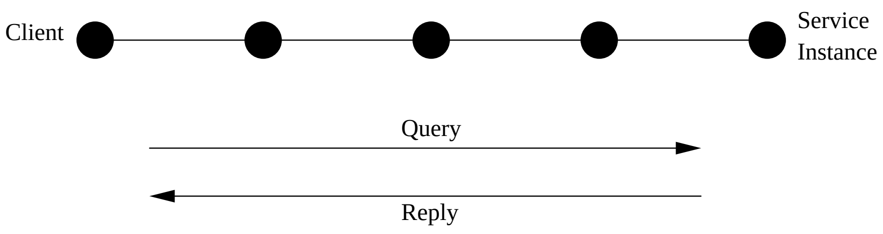

Figure 5.1: Long Chain

- **Number of devices.**
  - 4 Laptops IBM R32
  - 1 Laptop IBM T42 (not working properly)
  - 1 Desktop Computer
- **Software Tools.** The code was tested using the valgrind [13] debugging tool (version 2.2.0). The subversion [14] concurrent versions tool (version 1.0.8) was used for archiving the source code.

5.2 Functional Tests

If possible (up to 3 devices in a row), the tests were performed with real distances between the devices. For the bigger scenarios, the address filter tool (described in appendix B.5) was used. In the following, the most important tests are described:

- **Long Chain**

  Figure 5.1 shows the scenario with all devices in a row. The client sends a query message over four hops to a service instance and waits for a reply message. This scenario works without problems with 5 devices in a row using the MAC filter tool as well as with a chain of 3 devices with real distances.

- **Failover**

  Another scenario is shown in figure 5.2: when a relaying node leaves the network, the other nodes must adapt to the new topology. A query message is sent from the client (node 1) to the service instance (node 5) using the relaying nodes 2 and 3. Then, node 3 leaves the network. Thus, a subsequent query message to the same service instance needs to be redirected using node 4.

  The principle functionality of this test works fine but the actualization of the routing table needs too much time (waiting until the old neighbor times out and then set the new forwarding address). In future, this

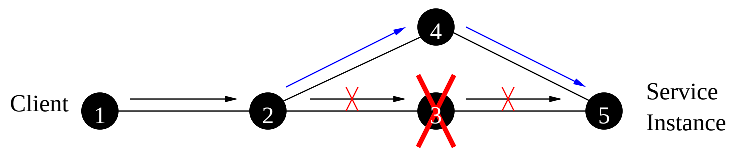

Figure 5.2: Failover

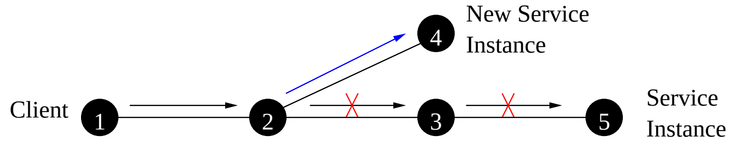

Figure 5.3: Mobility

can be improved by implementing the following: if the query is sent to a neighbor that is not in range any more, the wireless interface should answer with a *transmission fail message*. As soon as this message is received, the protocol can remove the disappeared neighbor from the neighbor list, set a new forwarding address for the corresponding service type and immediately send the query message again to another device.

- **Mobility**

  In scenario *mobility* (see figure 5.3), a service instance is replaced by another that just joined the network. The client (node 1) sends a query message to the service instance (node 5). At a later time, node 4 providing the same service type as node 5 (with the same capacity) joins the network and is closer to the client. Therefore, a subsequent query message from the client will be forwarded to device 4 instead of device 5. This scenario was tested successfully.

The scenarios above represent the most important test cases that can be arranged with 5 devices. During the implementation, there was ongoing testing necessary to see if the code works correctly. Furthermore, the scenarios above just demonstrate test cases with query and reply messages. Of course, also the building of the potential fields and the routing tables with advertisement and neighbor exchange messages was tested continuously but is not described in detail.

Chapter 6

Conclusion and Future Work

The goal of this master's thesis was to implement and test the MAgNETic routing protocol in C under Linux. The main tasks and their realization were the followings:

- **MAgNETic routing protocol design.**

  All necessary protocol messages and the MAgNETic protocol header are designed according to the model in the papers [3] and [4]. The first part (advertisement and neighbor exchange messages to build up the potential fields and the routing tables), described in detail in [3], was realized. No exact description existed for the second part (query and reply messages to search for a service and to send data). The complete design principles for this part were developed.

- **Implementation of the designed protocol.**

  The major part of the work was the implementation of the MAgNETic protocol. The basic concepts are the use of raw sockets for network access and to use different threads to guarantee a non blocking program execution. Especially because of the several threads, debugging was quite a challenging task. All applications and tools that were necessary to use and test the protocol (for example a tool to read out all intern data structures or a ping application to send a query message and wait for a reply) had to be developed as well.

  The basic functionality is implemented, but some tasks remain for future work: collecting advertisements in a node and forward them together to reduce the control traffic overhead in the network (described in [3]) is not yet implemented. Another open topic is the implementation of the distribution range limitation for reply service types (see section 3.4.4). An important part of future work is to define and implement an API for applications representing a service instance.

Currently, all provided service types are read in from a file. The MAgNETic protocol handles incoming query messages by automatically sending reply messages.

- **Validation with functional tests.**

  The implementation was tested using different hardware and software components. All relevant scenarios like failover or mobility were arranged. These tests demonstrate that the implementation works correctly and that the MAgNETic routing approach suits the requirements of the real environment. However, with a testing environment of five devices, it was not possible to perform complete evaluation tests.

  A problem that occurred was that the implementation did not work using a wireless LAN adapter based on an atheros chipset (see section 5.1). Sending an unicast frame to another device results in endless hardware interrupts and causes a crash of the entire computer. However, this is a minor issue and can easily be handled by using a wireless LAN adapter with a working chipset.

This implementation is a next step towards a testing environment to evaluate the performance of the MAgNETic service discovery and routing approach. It is topic of future work to set up a testbed with more nodes, for example using small devices like PDA's. The devices could be carried from test persons on their normal job over a long time. The code was not tested on PDAs, but we believe that porting it to PDAs should not be a major problem.

Testing with large testbeds causes a few new problems: the tests need to be analyzed. The logfiles of all devices provide plenty of data and should be analyzed automatically with some test tools. To arrange performance measurements, a time synchronization between the devices would be a huge advantage. However, this is not a simple task. For example, an approach for clock calibration [12] developed at the Computer Engineering and Networks Laboratory could be used.

Appendix A

Using the MAgNETic Protocol

This section shows how to install and run the MAgNETic protocol as well as the developed tools.

A.1 Installation

The code was developed for Linux. We tested the implementation with the Debian Linux kernels 2.4.24 and 2.6.10.

Installing the MAgNETic Protocol

Copy all files from the CD to an arbitrary directory. To compile, a GNU project C compiler (gcc) is required (tested versions 2.95.4 and 3.3.5). Change to the directory `source` and run the Makefile:

```
/home/user/magnetic/source# make
```

If desired, the debug mode and the logfiles to write can be set before compiling in file `./source/defines.h`.

Installing the Tools

To install the developed tools, run the script `compile_tools` in the directory `tools`:

```
/home/user/magnetic/tools# compile_tools
```

A.2 Starting

Setting up the Wireless LAN Interface

Set the wireless LAN interface to ad-hoc mode and select a cell number (essid). The cell number (for example 'testnet') must be the same for all participating devices. This can be done for example using the linux wireless tools (many wireless LAN drivers support these commands):

```
/home/user/magnetic# iwconfig wlan mode ad-hoc
/home/user/magnetic# iwconfig wlan essid testnet
/home/user/magnetic# ifconfig wlan up
```

Starting the MAgNETic Protocol

Start the MAgNETic protocol with the command `magnetic` in the directory `source` and select the network interface to use. The process must be started with root privileges.

```
/home/user/magnetic/source# magnetic wlan
```

Starting the Tools

Start the desired tool with the appropriate command (detailed description of all tools in appendix B):

```
/home/user/magnetic/tools# modify_services
/home/user/magnetic/tools# show_magnetic_data
/home/user/magnetic/tools# ping_service
/home/user/magnetic/tools# send_query
/home/user/magnetic/tools# reload_mac_filter
```

A.3 Important Files

All possible logging information is written to the following files:

```
./source/log/all_packets        // logs all packets
./source/log/corrupted_packets  // logs all wrong packets
./source/log/whole_packets      // logs the whole packets
./source/log/neighbors          // logs all neighbors
```

A list of all provided services and all data that arrived for these service instances can be found in:

```
./services/services.cfg         // all provided services
                                //   and capacity
./services/received_data        // received data
```

Appendix B

Developed Tools and Applications

This section describes all developed tools and applications. Appendix A describes how to install the tools. Before starting any tool or application, check that the MAgNETic protocol must be running at the node. The tools must be started with root privileges.

B.1 Management of Provided Services

Purpose

The tool to manage the provided services is used to add, change or delete provided service instances. The provided service instances are written to the file ./services/services.cfg (service type and capacity) and delivered to the MAgNETic protocol with every change. For these service instances, no application is running in the background. The MAgNETic protocol currently simulates an application that reacts to query messages and sends a reply.

Usage

Command to start the tool to manage provided services:

```
/home/user/magnetic/tools# modify_services
```

Select what to do next:

```
Select Action: 1: Show Provided Services
               2: Add a Service Instance
               3: Delete a Service Instance
               4: Modify the Capacity of a Service Instance
               9: Exit
```

B.2 Print Intern MAgNETic Protocol Data

Purpose

The tool to print intern MAgNETic data is basically used for debugging and testing. It allows to see intern data (routing table, neighbor list, etc.) at runtime. To reconstruct what happened before, have a look at the logfiles.

Usage

Command to start the printout tool:

```
/home/user/magnetic/tools# show_magnetic_data
```

Select the data to print:

```
Select Action: 1: Print Routing Table
               2: Print all Advertisements
               3: Print provided Services List
               4: Print Neighbour List
               5: Print Statistics
               9: Exit
```

B.3 Ping Application

Purpose

The ping application sends a query message of type 2 (acknowledged, see section 3.4) to a service instance of the desired service type and waits for a reply message. Consider that the subsequent query messages need not to be answered by the same service instance. Currently, the MAgNETic protocol answers all queries with query type acknowledged with sending back the payload data. In further implementations, the applications acting as service instances should implement the ping reply themselves.

Usage

Command to start the ping application:

```
/home/user/magnetic/tools# ping_service
```

Then select the service type to ping. Magnetic sends query packets to the optimal service instance of the given service type until you stop the application (use CTRL-C) or `MAX_PACKETS` (defined in the file ping_service.h) packets are sent.

B.4 Send Query Application

Purpose

The send query application sends a query message type 1 (unconfirmed, see section 3.4) to a service instance of the desired service type. The service instance stores the received data in the data file.

Usage

Command to start the send query application:

```
/home/user/magnetic/tools# send_query
```

Then select the service type to search for and the data to append to the query (currently 4 bytes of data are appended).

B.5 MAC Address Filter Tool

Purpose

For testing and debugging, it is an advantage to select the packets that a device should receive. With the MAC address filter tool, the packets with a specific source address are dropped.

Usage

Write the MAC addresses of the incoming packets to filter into the file ./tools/denied_mac.cfg, for example:

```
00 20 e0 8e dd 75
00 20 e0 8e dc e5
00 20 e0 8e dd 89
```

Check that the option `MAC_FILTER` in the file `./source/defines.h` is set to allow the MAC address filter option. Then start the program

```
/home/user/magnetic/tools# reload_mac_filter
```

to read the new addresses to filter with the MAgNETic protocol. From now on, all incoming packets from a device with one of the specified source addresses will be dropped. To disable the filtering of packets, simply delete the file `./tools/denied_mac.cfg` and start `./reload_mac_filter` again.

Bibliography

[1] Swiss Federal Institute of Technology Zurich
`http://www.ethz.ch`, April 2005.

[2] Computer Engineering and Networks Laboratory
`http://www.tik.ee.ethz.ch`, April 2005.

[3] Vincent Lenders, Martin May, and Bernhard Plattner. *Service Discovery in Mobile Ad Hoc Networks: A Field Theoretic Approach.* To appear in the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), Taormina, Italy, June 2005.

[4] Vincent Lenders, Martin May, and Bernhard Plattner. *Towards a New Communication Paradigm for Mobile Ad Hoc Networks.* TIK Report 203, ETH Zurich, Switzerland, August 2004.

[5] Jon Postel. DoD Standard Internet Protocol. IETF RFC 760, January 1980.

[6] Xiang Zeng, Rajive Bagrodia, and Mariio Gerla. "GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98)*, Banff, Alberta, Canada, May 1998.

[7] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. "Ad Hoc On-Demand Distance Vector (AODV) Routing," IETF Internet Draft, draft-ietf-manet-aodv-12.txt, November 2002.

[8] Thomas Clausen and Philippe Jacquet. "Optimized Link State Routing Protocol," IETF Internet Draft, draft-ietf-manet-olsr-11.txt, July 2003.

[9] David B. Johnson, David A. Maltz, Yih-Chun Hu, and Jorjeta G. Jetcheva. "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)," IETF Internet Draft, draft-ietf-manet-dsr-09.txt, April 2003.

[10] Ian D. Chakeres, Elizabeth M. Royer, and Charles E. Perkins. "Dynamic MANET On-demand Routing Protocol," IETF Internet Draft, draft-ietf-manet-dymo-00.txt, Februar 2005

[11] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming: The Socket Networking API.* Pearson Education, Inc, Boston, 2004.

[12] Philipp Blum and Georg Dickmann. *Precise Dely Measurements in Wired and Wireless Local Area Networks.* Tik Report 175, ETH Zurich, Switzerland, July 2003.

[13] Valgrind: x86 Linux Debugging Tool.
`http://www.valgrind.org`, April 2005.

[14] Subversion: Version Control System.
`http://subversion.tigris.org`, April 2005.