Manuel Graber

Distributed Service Discovery and Session Initiation using SIP for MANETs

Master Thesis
January 5, 2005 – July 4, 2005

Supervising Assistant: Patrick Stüdi
Supervising Professor: Prof. Gustavo Alonso
Abstract

The *Session Initiation Protocol (SIP)* is a signalling protocol for Internet conferencing, telephony, presence, event notification and instant messaging. Unfortunately, SIP cannot be used in Mobile Ad hoc Networks (MANETs) directly because it relies on centralised services which conflicts with the notion of decentralisation usually associated with MANETs.

An analysis of the existing SIP architecture and their problems in MANETs leads to the conclusion that the problem of SIP in MANETs can be mapped to the more generic problem of *Distributed Service Discovery (DSD)*. Based on a solution for DSD, the centralised part of the SIP infrastructure can be replaced by a distributed counterpart.

In order to tackle the DSD problem different existing approaches are studied. The fully distributed approach to DSD is found to be the most promising one. Analogously to MANET routing protocols, DSD using a fully distributed approach can be done in a reactive manner, in a proactive manner or using a combination of both. The evaluation of different strategies leads to the result that the reactive one is generally a good option for DSD in MANETs, particularly concerning performance and complexity.

In a second step, a very flexible, efficient and generic design for a DSD system in MANETs is proposed. It utilises messages sent by the routing protocol, e.g. AODV, to transport the messages needed for DSD. The design is flexible enough to support different routing protocols and different strategies of service discovery. This allows to tailor the DSD system to different kinds of MANETs.

Finally, an existing implementation of the centralised part of the traditional SIP infrastructure, the *SIP Proxy Server*, was adapted to use the DSD subsystem to discover other SIP users. This leads to a solution for SIP in MANETs that can be used with existing *out-of-the-box SIP User Agents*.

The proposed design was implemented for the GNU/Linux operating system using AODV as routing protocol and was tested in a multi-hop setup.
Contents

1 Introduction 5
  1.1 Motivation ........................................... 5
  1.2 Goal of the Thesis .................................... 5
  1.3 Session Initiation Protocol (SIP) ....................... 7
    1.3.1 Who is Who? ..................................... 7
    1.3.2 The Registration .................................. 7
    1.3.3 The Call ........................................... 9
    1.3.4 Other SIP Methods ................................. 9

2 Problem Analysis and Solution Concept 10
  2.1 Centralised Servers in MANETs ......................... 10
  2.2 Service Discovery Problem ............................. 12
  2.3 Distributed Service Discovery (DSD) ................. 12
    2.3.1 Approaches to Distributed Service Discovery (DSD) in MANETs ......................... 13
  2.4 Semantics of DSD ..................................... 15
    2.4.1 Service Descriptor ............................... 15
    2.4.2 Service Payload .................................. 15
    2.4.3 Application Level Interface ..................... 16
    2.4.4 Equality of Services ............................. 17
  2.5 Distributed SIP Architecture ........................ 17
  2.6 Transport Layer Issues ................................ 18
    2.6.1 Using specialised service discovery messages .... 18
    2.6.2 Using Routing Messages for Service Discovery 19
  2.7 Related Work .......................................... 21
    2.7.1 Architecture Comparison .......................... 21
    2.7.2 Service Discovery in OLSR MANETs ............... 22
    2.7.3 Instant Messaging and Presence Services in Wireless Ad Hoc Networks ................. 22
    2.7.4 Consistency of Service Discovery in MANETs .... 22

3 Design 23
  3.1 Requirements ......................................... 23
  3.2 Coarse grained partition of Distributed Service Discovery .... 24
  3.3 Message Format ....................................... 24
  3.4 Message Types ....................................... 26
  3.5 Data Types ........................................... 26
  3.6 Transport Layer ....................................... 26
3.7 Message Dispatcher .............................................. 27
   3.7.1 Socket Listener ........................................... 28
   3.7.2 Transport Handler ....................................... 31
3.8 Service Handler ................................................ 31
3.9 Design of the SIP Service Handler (SIPHoc) .................. 31

4 Implementation ......................................................... 33
   4.1 Development Environment .................................... 33
      4.1.1 Hardware ............................................... 33
      4.1.2 Software .............................................. 34
      4.1.3 Programming Languages .............................. 35
   4.2 Common Utility Classes ...................................... 35
      4.2.1 Threading .............................................. 35
      4.2.2 TLV Classes ........................................... 36
      4.2.3 Miscellaneous ......................................... 38
   4.3 Distributed Service Discovery System ....................... 41
      4.3.1 Generic Part of the Transport Layer .................. 41
      4.3.2 AODV-specific Part of the Transport Layer .......... 46
      4.3.3 Message Dispatcher .................................... 51
   4.4 SIP Proxy Server using DSD ................................ 54
      4.4.1 Porting TLV to Java ................................... 55
      4.4.2 Integrating JAIN-SIP Proxy to DSD ................... 55

5 Debugging and Testing ................................................ 58
   5.1 Unit Tests .................................................. 58
   5.2 Integration Testing ......................................... 58
      5.2.1 Transport Handler Test ................................ 58
      5.2.2 Socket Listener Test .................................. 59
   5.3 Memory Management ......................................... 60

6 Conclusion and Future Work ........................................... 61
   6.1 Summary .................................................... 61
      6.1.1 Distributed Service Discovery ......................... 61
      6.1.2 SIPHoc: SIP for MANETs ............................... 61
   6.2 Limitations .................................................. 62
   6.3 Future Development ......................................... 62

A How to use ............................................................ 63
   A.1 Platform Requirements ....................................... 63
   A.2 Building Process ........................................... 64
   A.3 Configuration ................................................. 64
      A.3.1 Configuration of the DSD System ....................... 64
      A.3.2 Configuration of the SIP Proxy Server ................. 65
      A.3.3 Configuration of the SIP User Agent .................. 66
   A.4 Running and using ........................................... 66
   A.5 Developing new Service Handlers ............................ 67
   A.6 Developing new Routing Handlers ........................... 67
   A.7 Developing new Strategies .................................. 67

B Configuration Files ................................................ 69
CONTENTS

C Data Types 71
D Message Types 73
List of Figures 75
List of Tables 77
List of Listings 78
This master thesis was done in parallel with the master thesis [1]. To use synergies, some ideas as well as a part of the implementation code and its documentation were shared.
Chapter 1

Introduction

1.1 Motivation

The Session Initiation Protocol (SIP) is a peer-to-peer, multimedia signalling protocol developed in the IETF. SIP is ASCII-based, resembling HTTP, and reuses existing IP protocols (DNS, SDP, etc.) to provide media setup and teardown. Since its first publication in 1999, SIP has generated a high level of interest in the Voice-over-IP (VoIP) industry, and many people believe that SIP will become the de facto standard protocol for future voice networks. SIP is revolutionising telecom and enterprise network infrastructure and services as it represents a shift from proprietary switching technologies to a computing-centric IP based approach.

The telephony world has evolved from an analog telephone network to a digital circuit-switched network over the last couple of decades. Recently, we are riding a new wave to move from circuit-switched to packet-switched technology for carrying both voice and data over IP networks. This has led to the emergence of VoIP technology, where reduced long distance tariffs, eliminated access charges and added-value services played a major role in promoting the technology. While most work done so far concentrates on wired, infrastructure based networks (Internet, enterprise networks) the question arises whether VoIP could also be used in wireless, non-infrastructure environments, like e.g. Mobile Ad Hoc Networks (MANETs). Assuming a university campus, one could imagine phone calls to be routed through the ad hoc network whenever available, which is free of costs (see Figure 1.1).

A problem is that SIP relies on a centralised infrastructure (see Figure 1.2). Unfortunately MANETs are inherently decentralised, mobile and error-prone in transmission. This means that to use SIP in MANETs a replacement for all centralised entities in the SIP architecture has to be found.

1.2 Goal of the Thesis

The goal of this master thesis is to design and implement a working SIP solution for MANETs. This should be done with respect to the following criteria:
1.2 Goal of the Thesis

Figure 1.1: Transition from infrastructure based telephone networks to infrastructureless telephone networks.

Figure 1.2: Overview over all actors of SIP.
1.3 Session Initiation Protocol (SIP)

SIP can be used for various applications like multimedia conferencing or instant messaging.

The following sections give a short overview over the most important parts of the SIP. More detailed information can be found in [2, 3]. More information about SIP and mobility can be found in [4].

1.3.1 Who is Who?

A SIP user is identified using a *SIP Uniform Resource Indicator (URI)* which, in case of telephony, replaces the usual telephone number and looks as follows:

```
sip: username@domainname
```

An example of a SIP URI would be:

```
sip:testuser@mydomain.com
```

SIP messages are in general sent using UDP or TCP. There exists also a specification for *Secure SIP* that uses TLS over TCP for communication. In this case the leading *SIP* in the SIP URI is replaced by *SIPS*. Secure SIP is not further considered in this thesis.

In general there are four different components that play a role in SIP. They are explained in table 1.1. Besides these four entities, SIP uses standard services of an IP network, e.g. DNS.

1.3.2 The Registration

The first step after a SIP user agent has been started is to register at the SIP registrar responsible for its domain. In order to do this it sends a REGISTER request and receives a corresponding answer, see Figure 1.3.
1.3 Session Initiation Protocol (SIP)

<table>
<thead>
<tr>
<th>Component</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIP User Agent</td>
<td>User Agents are the endpoints of a SIP session. They can be implemented in software as well as in hardware. In case of Internet telephony, they offer a more or less classical telephony interface to the user (receiver, number pad, ...).</td>
</tr>
<tr>
<td>SIP Proxy Server</td>
<td>A SIP Proxy Server is responsible for handling requests received from a User Agent or another SIP Proxy Server. For example, session establishment requests are handled by a SIP Proxy Server.</td>
</tr>
<tr>
<td>SIP Location Service</td>
<td>The SIP Location Service is a kind of database that holds the mappings between SIP User Name (SIP URI) and IP address. It serves the SIP Registrar to store and look up these mappings. The SIP Location Service is often collocated with the SIP Registrar and/or the SIP Proxy Server.</td>
</tr>
<tr>
<td>SIP Registrar</td>
<td>A SIP Registrar holds all the information about all User Agents of a domain that are ready to receive or make phone calls. SIP Registrars are mostly collocated with SIP Proxy Servers on the same node or even in the same process. Typically, there is one SIP Proxy Server / SIP Registrar per domain.</td>
</tr>
</tbody>
</table>

Table 1.1: The four most important components of SIP.

Figure 1.3: SIP Register Request.
1.3 Session Initiation Protocol (SIP)

1.3.3 The Call

The actual SIP call is done sending an INVITE message to the own proxy server—in this case called outbound proxy server (see Figure 1.4). The outbound proxy server analyses the request and tries to locate the SIP proxy server responsible for the callee’s domain. This is done by sending a DNS SRV request to the DNS server [5, 6]. The DNS server answers with the corresponding IP address or hostname. The outbound proxy server finally forwards the request to the callee’s inbound proxy server. The inbound proxy server consults its registrar to find out whether the callee is currently registered and therefore ready to answer a call or not.

If the callee is registered, the inbound proxy server gets the current IP address from the SIP location service and forwards the INVITE message to the callee’s user agent.

This trapezoid message path is only used for SIP control messages. The actual traffic (voice, multimedia, ...) is sent directly from the caller’s user agent to the callee’s user agent.

1.3.4 Other SIP Methods

The SIP methods explained above (REGISTER, INVITE) are part of the official SIP specification (RFC 3261) [2]. Besides these methods, several additional have been specified. Especially Instant Messaging applications take use of these additional methods. See [7, 8, 9, 10, 11, 12] for details.
Chapter 2

Problem Analysis and Solution Concept

In this chapter the problems of existing SIP solutions in MANETs are analysed and possible solutions are discussed. Finally, an overview over related work is given.

2.1 Centralised Servers in MANETs

When we want to use existing SIP solutions in MANETs we encounter some problems. There is one main problem to deal with: it is a really bad option to have centralised servers in MANETs for various reasons. They are one of the classical problems in MANETs. What happens for example if the server moves out of range or just loses its link to the rest of the network? These situations can occur often in MANETs and may not influence the normal operation of the network. Consequently, centralised servers have to be avoided by all means in MANETs. The implication of this is that most known infrastructure network services are not available in MANETs because they rely on centralised servers.

SIP uses DNS SRV entries to get information about the SIP proxy server responsible for a given domain name. This is impossible in MANETs since in general no DNS service is available. Therefore, the absence of DNS is a problem for existing SIP solutions.

As seen in section 1.3, there are the following centralised servers in the traditional SIP setup:

- DNS Server
- SIP Registrar
- SIP Location Service
- SIP Proxy Server

To get a working SIP infrastructure in MANETs it is necessary to replace these centralised servers by some kind of distributed mechanism that is adequate for MANETs. This does not mean that all of these servers have to be replaced
with a MANET capable counterpart. It is possible that a solution for the problem consists in replacing some or all of these servers by one completely new approach that is tailored for MANETs.

If we take a closer look at the existing SIP architecture, we see that the main problem we got is resolving the domain name part of a SIP URI to a numerical IP address, see Figure 2.1. This is done going through the following steps:

1. The caller’s outbound proxy server issues a DNS SRV request to find the callee’s inbound proxy server.
2. The callee’s inbound proxy server asks its SIP registrar or its SIP location service respectively whether the user in question is registered and what IP is assigned to him.
3. Finally the SIP message is forwarded to the resolved IP address.

The first step is only necessary if the caller’s proxy server is not identical to the callee’s proxy server. This means that we do not need a DNS service if we use the same proxy server for all nodes in the MANET. Furthermore, if we have the same proxy server for all nodes then we also use the same SIP registrar and consequently the same SIP location service for the whole MANET.

Did we solve the problem? No, unfortunately not. Using one central server in a MANET is actually the very thing we want to avoid. But it is clear now that the problem can be solved without accessing a DNS service.

So what do we have so far: the problem can be solved if all nodes use the same SIP proxy server, but we cannot use one single centralised server in a MANET. The question is now how to replace this centralised SIP proxy server by a distributed counterpart.

The first step towards a solution to replace the centralised SIP proxy server is to take a closer look at the core problem: the mapping from a SIP URI containing a symbolic domain name to a SIP URI containing the real IP address of the user. This mapping problem can be transformed into a well discussed problem; what we really need is Service Discovery in MANETs. Some node has a SIP user agent running and offers therefore the service to accept SIP sessions for the currently registered user identified by a SIP URI. Another node tries to discover on which node a user is registered or, in other words, on which node this specific service is offered. This is a classical service discovery problem. A little bit special about it is that we cannot rely on centralised infrastructure to solve this problem.

We first reduced our rather complicated SIP scenario to the well known problem of replacing a centralised server in a MANET by a distributed counterpart.
2.2 Service Discovery Problem

Service discovery is the problem of having a service descriptor and looking for a corresponding service information. In case of SIP, the service descriptor would be a SIP URI containing a symbolic host address and the service information would be a SIP URI containing an IP address. DNS is a service discovery problem, too. In this case the service descriptor is a full qualified domain name and the service information is the corresponding IP address. Another scenario are location based service look-ups like “Where is the next hotel?” (see Figure 2.2).

Therefore, it makes sense to solve the more generic service discovery problem and use it for SIP rather than going for a SIP-specific solution.

The usual solutions follow an approach that is based somewhere between the following two extremes:

1. Service information is offered by one central node called service coordinator.
2. Service information is distributed among all nodes in the network. This means every node is its own service coordinator.

In general, a node that wants to offer a service registers its service at a service coordinator. This process is also called publishing a service. A node that is interested in a service sends a request to a service coordinator and gets information about the service back, e.g. the IP address of the node where the service is available. This is called looking up a service.

The problem with available service discovery mechanisms for infrastructure based networks, e.g. JINI [13] or SLP [14], is that they are usually not designed for MANETs. They either rely on centralised structures or do not perform well in such environments.

2.3 Distributed Service Discovery (DSD)

To solve the service discovery problem in MANETs an architecture is needed that fulfils all the constraints that hold in MANETs and offers a good perfor-
2.3 Distributed Service Discovery (DSD)

Besides the use of centralised servers (see Figure 2.3(a)) which was shown to be inadequate, there are basically two further approaches to solve the problem. See also [15].

### Overlay Backbone / Hybrid Approach

The idea behind this approach is to elect a number of nodes that work as service coordinators rather than one single node, see Figure 2.3(b). State is synchronised between all of these nodes. As a result of this, each node in the network has access to at least one service coordinator. This approach is feasible for MANETs. If a service coordinator leaves the network, no state information is lost.

---

**Figure 2.3:** Comparison of three approaches to distributed service discovery.

(a) Centralised approach to replace central servers.

(b) Virtual Backbone approach for replacing centralised servers.

(c) Fully distributed approach for replacing centralised servers.
is lost and another node can take over its place. Availability in this approach is good since at all times there are some service coordinators in the virtual backbone.

The drawbacks using the approach are:

- It is rather a complex problem to construct a virtual backbone.
- The virtual backbone has to be maintained. This is also a complex problem and produces extra traffic—especially in case of node mobility.

**Fully Distributed Approach**

The fully distributed approach describes the concept that either all nodes are service coordinators or no node is service coordinator, see Figure 2.3(c). Basically, two different strategies can be distinguished:

**Strategy**  The term *strategy* describes the way service information is exchanged in a fully distributed approach to distributed service discovery.

- **Zero knowledge strategy**

  This means that a node just knows about the services it provides itself. If a node needs to discover a service that is located somewhere else, it broadcasts a service request and gets the answer from the node that provides the service in question. This strategy is also referred to as query-based or lazy strategy. Or with other words: no node is actually a service coordinator (apart from for the local services).

- **Global knowledge strategy**

  Using this strategy all service information is spread throughout the network proactively. The result is that all nodes have a global knowledge about all services available in the whole network. Or with other words: all nodes are their own service coordinators.

Another option is a combination of these two strategies. For example we could spread some service information proactively while other services are accessed in a query-based manner. Or, it is thinkable to spread service information proactively only when this can be done for free, e.g. together with messages that are sent anyway.

Which one of the strategies offers the better performance is heavily dependent on the MANET itself. Actually it is closely related to the question of which routing protocol performs best in MANETs, since routing protocols also use the two strategies mentioned. This is discussed more deeply in section 2.6.

But the complexity of such a system is low. It is not necessary to maintain a virtual backbone or to elect some nodes as service coordinators. Availability is good because in case of the zero knowledge strategy all requests for a service are answered by the node that offers the service. So, it can be discovered as long as it is available. In case of the global knowledge strategy service requests are answered locally. This means that a request is answered in every case. There is currently much work done in this area, see section 2.7.
2.4 Semantics of DSD

Chosen Solution
From the implementation point of view, the solution with the lowest possible complexity is preferable. In this case, the fully distributed approach would be the best choice.

But does this solution also perform well? The answer to this question can be found in [16]. The conclusion of this paper is that a distributed query-based approach outperforms a hybrid approach in reactively routed MANETs and that it also seems to be preferable in proactively routed MANETs. Consequently, there is no reason to go for the more complex hybrid approach.

2.4 Semantics of DSD

What now remains, after having chosen to go for the distributed query-based approach, is to define the syntax and semantics of what service discovery actually means.

Basically, a service is defined by a service descriptor and a service information which is also called service payload. Important is to remain as generic as possible in order that there are as less restrictions as possible to applications that want to do service discovery.

2.4.1 Service Descriptor

For implementation reasons, a service descriptor needs to be something that can be compared to another service descriptor and it would make sense when service descriptors could be ordered. This allows to use service descriptors as entries in data structures.

Another requirement is that the service descriptor has to be unique. Imagine a scenario where two different types of applications use service discovery, e.g. DNS and SIP. The DNS applications do not need to know that there are also SIP applications that use service discovery. If now by hazard a SIP application chooses the same descriptor as the DNS application, a service look-up of the SIP application may return a DNS service and vice versa. This can happen even if the service descriptors are unique in the scope of a type of applications, i.e. all DNS service descriptors are unique and all SIP service descriptors are unique.

A simple string of characters would meet these requirements, besides the last one. The solution is to split the service descriptor into two parts, a service class and a service identifier, see Figure 2.4.

The service class determines to what type of application the service belongs, e.g. there is a service class for DNS and one for SIP. Now it is in the domain of the application to handle the uniqueness issue or not. There might be applications where it makes sense to use non-unique identifiers.

As service class e.g. an integer can be used. As actual service identifier a string of characters is suitable.

2.4.2 Service Payload

For the service payload there are no constraints to be fulfilled. It is not necessary that a service payload can be compared neither ordered. Uniqueness is also not
2.4 Semantics of DSD

2.4.3 Application Level Interface

The interface to the application can be held very slim. To do service discovery only two primitives are necessary: making a service available and searching for a service.

Besides these two primitives a control interface is needed. The DSD system must know the responsible application for each service class. This is ensured by a third primitive called registration. What follows are the definitions of these primitives.

Publishing a Service

Publishing a service means making a service discoverable. If an application that provides a service wants that it can be discovered by other applications using DSD, it issues a publish for this service. A service is discoverable from the point in time it is published. If the service is no longer available the application issues a negative publish message to unpublish the service.

Looking up a Service

Looking up a service means try to discover a service. If an application needs a specific service it issues a service look-up to the DSD system. It is able to discover all services that have been published by any node in the DSD system.

Registering at the DSD System

An application that wants to use the DSD system must first register itself. Registrations are carried out per service class. This is necessary because the DSD system needs to know which application is responsible for services of which class in order to deliver the results of service look-ups to the right application.

Figure 2.4: Composition of a service descriptor.

an issue for the service payload. So, a natural solution is to use a byte buffer as service payload.
2.4.4 Equality of Services

Two services are considered to be equal if service class as well as service identifier are equal. Consequently, a service look-up request contains a service class and a service identifier. A service that matches to this service look-up request contains the same service class, the same service identifier plus a specific service payload.

2.5 Distributed SIP Architecture

One of the goals of a distributed SIP architecture is to remain compatible with existing standard compliant SIP components like user agents and proxies.

To achieve this goal it is desirable to make as little changes to the existing architecture as possible. As stated above, we can use distributed service discovery to solve the SIP URI mapping problem. The question is now how to integrate this into the existing SIP architecture.

If we take a closer look at the SIP architecture, we can see that this mapping is done in the SIP location service component. A SIP location service is usually, together with a SIP registrar, a part of a SIP proxy, see Figure 2.5.

An elegant solution to the problem is to take an existing SIP location service and make it use distributed service discovery instead of a local database to process mapping requests. What we get is a Distributed SIP Location Service, see Figure 2.6. In this case, all other parts of the SIP architecture can remain unchanged.

When we have a distributed location service, the question arises how to access it. In the normal SIP architecture the SIP location service is accessed by a SIP registrar or by a SIP proxy. A straight-forward solution is to have an own SIP proxy server, including registrar and location service, for every node. This leads us to a similar setup as the one shown in Figure 1.4. The differences are (see Figure 2.7):

- The SIP location service at every SIP proxy server is replaced by a distributed SIP location service
- The DNS server is no longer needed, since it is finally just used to find the remote SIP proxy server or the remote SIP location service, respectively.
- A SIP proxy server is running on every node in the network and is just responsible for this very node.
2.6 Transport Layer Issues

Another important issue is the choice of a transport method for the service discovery messages. There are basically two options: send own service discovery messages to the network or piggyback service discovery information to messages that are sent either way.

The question however is: what messages are sent either way to the network? Certainly, routing messages fall into this class. Apart from that, routing and service discovery are closely related. If a node issues a service look-up request, it normally wants to access this service. This means that it needs a route to the node that offers the service.

The following sections describe the two options in detail.

2.6.1 Using specialised service discovery messages

Sending extra messages for service discovery is the easiest solution. This can be implemented straight-forward without a big effort. The drawbacks of this method are:

- Puts extra load on the network.
2.6 Transport Layer Issues

Service discovery messages often need to be broadcasted throughout
the network. Hence, sending extra messages produces a considerable addi-
tional load on the network.

- Induces extra routing messages.

When using extra messages to spread service discovery information
through the network, this induces a considerable routing overhead. If
service information messages are broadcasted no routes to the nodes that
offer the services are established during the service look-up phase; it has to
be done later when someone wants to use a service. This can possibly be
avoided by piggybacking the service information to routing messages. On
the other hand, if service information is unicasted, each unicast triggers a
route to be established.

- Sending of messages in a straight forward manner is not optimised.

There is much optimisation potential when sending messages in a
MANET, especially for broadcasts. Generally routing protocols send mes-
sages in a highly optimised manner. They use e.g. expanding ring search,
binary exponential backoffs or optimised flooding schemes.

These drawbacks lead to the conclusion that it is not the best option to send
extra service discovery messages.

2.6.2 Using Routing Messages for Service Discovery

Piggybacking service discovery information to routing messages offers the pos-
sibility to adjust the lifecycle of service discovery information to the routing
protocol. This means that existing optimisations like optimised flooding or
expanding ring search can be used for service discovery, too.
2.6 Transport Layer Issues

Another important point is that since service information is sent together with routing messages, routes are established together with service discovery. So, in the optimal case, no routing overhead is induced. The concept of including service discovery or other messages in routing messages was already discussed in the internet drafts [17, 18].

This brings up the question about the most appropriate routing algorithm for this scenario. The most common routing algorithms for MANETs can be separated into two classes: reactive algorithms and proactive algorithms.

Reactive Routing Algorithms

This sort of algorithms works in two steps. If a node needs a route to a destination, it broadcasts a route request message (RREQ). This route request is then answered either by the destination itself or by an intermediary node that knows a route to the destination. The answer is unicasted back in a route reply message (RREP). The most common member of this class is the Ad hoc On-Demand Distance Vector Routing (AODV) [19]. A widely-used implementation is AODV-UU. This implementation is done by the university of Uppsala and work is still in progress.

Proactive Routing Algorithms

Proactive routing algorithms establish routes before they are really requested. Routing information is exchanged proactively. A well-known member of this class of routing algorithms is the Optimised Link State Routing (OLSR) [20]. A common implementation was done in the master thesis [21].

From the performance point of view, there is no clear answer which kind of algorithm to use. There are network setups where a reactive algorithm performs best but there are also setups where proactive algorithms perform best. For a comparison between AODV, OLSR and another reactive routing protocol (Dynamic Source Routing (DSR)) see [22].

This means a generic service discovery solution for MANETs should not rely on a specific routing protocol. It is desirable that a service discovery solution is easily adaptable to various routing protocols.

Since we follow the distributed query-based approach as discussed in section 2.3, it seems to be a good choice to use a reactive routing protocol for the prototype implementation. The main reason for this is that the kind how reactive protocols work can be mapped directly to the operation mode of the query-based approach for distributed service discovery. If a route is needed the reactive routing protocol issues a route request message and gets a route reply back from the destination. This is actually the same as what a node does in a query-based service look-up: it sends a service look-up message and gets back the service information. Correspondingly, the query-based approach to service discovery can be perfectly integrated with a reactive routing protocol.

Nevertheless, this query-based approach should be encapsulated in the prototype implementation to make it possible to exchange this query-based strategy for another strategy, e.g. for a global knowledge strategy. The reason is that there might be situations or environments where such a strategy performs better or is the only feasible. This might be the case especially in combination with proactive routing protocols.
2.7 Related Work

There have been published a lot of interesting papers concerning service discovery in MANETs or the closely related topic of name resolution in MANETs. In the following sections we will refer to some of them.

2.7.1 Architecture Comparison


It proposes the use of routing messages to piggyback service discovery information. The conclusion of the authors is that the fully distributed approach
outperforms the hybrid approach for reactively routed MANETs.

2.7.2 Service Discovery in OLSR MANETs

A paper on service discovery for SIP applications in proactively routed MANETs is [23]. It describes how the OLSR routing protocol can be used to transport service discovery messages. Two different schemes are proposed:

- **Automatic Service Advertising (ASA)**
  
  Using this scheme a service provider periodically announces all its services to the whole network using OLSR’s optimised flooding facility (MPR forwarding).

- **Client Query and Server Advertising (CQSA)**
  
  This scheme provides the additional ability to the client to explicitly query for services. This can be used to prevent a delay if for some reason no server advertisement was heard by the client.

The authors compare these two schemes to the normal OLSR background traffic in a simulated environment. The conclusion is that the traffic overhead induced by the proposed schemes is not significant compared to the normal OLSR background traffic.

2.7.3 Instant Messaging and Presence Services in Wireless Ad Hoc Networks

In [24] the instant messaging and presence extensions to SIP and their use in wireless ad hoc networks are discussed. For delivering SIP messages the authors propose a cluster based routing protocol. This is an option because using the SIP MESSAGE method also the actual session payload (instant messaging messages) and not only the session control messages are sent using SIP proxy servers. For other SIP applications, such as telephony, the actual payload is sent independently from the SIP proxy servers.

2.7.4 Consistency of Service Discovery in MANETs

In [25] a service discovery solution that uses AODV routing messages as transport for service discovery information is presented. The authors propose an interface for the routing daemon, so that the service discovery application can control the routing messages sent and received. The impact of caching is examined and proposals are made on how to maintain cache consistency. A simulation of the proposed system finally leads to the conclusion that a service discovery can be integrated efficiently with route discovery.
Chapter 3

Design

This chapter describes the design of the DSD system and of SIPHoc in detail. The separation into several parts and layers is explained as well as the structure of these parts.

Moreover, the functionality of the parts is described and how they interact with each other. For a documentation on class or function level please refer to the separate API documentation, to the source code or chapter 4.

3.1 Requirements

The following requirements and constraints need to be fulfilled by the design of DSD and of SIPHoc:

• Performance

For performance reasons, as little messages as possible shall be sent over the network. It is important to take also into account the routing messages that are possibly induced by other messages. As seen in section 2.6, the use of existing routing messages is a promising approach to send service discovery messages.

• Different Transport Layers

Since it is desired to use existing routing messages as transport method, it is necessary that the transport layer is partly specific to the routing protocol used.

To offer support for as many environments as possible this routing algorithm specific part must be encapsulated. This makes it possible to implement transport layers for different routing protocols so that they can be plugged into the system.

• Different Strategies

As stated in section 2.3.1 there are different strategies for implementing DSD in a fully distributed manner. The prototype will implement a query-based strategy. To be as flexible as possible this strategy must be encapsulated in order that it can be exchanged by another one, e.g. a global knowledge strategy. This enables the use of arbitrary combinations of strategy and routing protocol (see section 2.6).
3.2 Coarse grained partition of Distributed Service Discovery

DSD is divided into two main parts (see Figure 3.1):

- **Transport Layer**
  
  The transport layer is responsible for getting routing messages from the IP stack and for sending messages back to the IP stack.
  
  It is also responsible for the low-level handling of routing messages. Consequently, it offers the functionality to put payload into messages and to read payload from the messages.

- **Message Dispatcher**
  
  The message dispatcher is responsible for the handling of service discovery messages. It gets messages from the application layer and from the transport layer and processes them appropriately.

The part of an application that enables this application to use service discovery functionality is called *service handler*. It acts as client to the DSD system and therefore sends requests to the message dispatcher and gets answers from there. Instead of being part of the application, the service handler may also be a wrapper for an existing application.

For the communication between all the parts of the architecture and most of all for the communication between the instances on all the different nodes a message format is needed.

### 3.3 Message Format

Requirements for the message format are:

- **Support for message types**
  
  It shall be possible to define message types. This offers the possibility to assign a notion of context to the message payload. If an integer is sent, it
3.3 Message Format

Figure 3.2: Example for using message types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;ABCDE&quot;</td>
</tr>
</tbody>
</table>

Figure 3.3: Example TLV message.

can be specified what the actual meaning of this integer is, see Figure 3.2. Furthermore this allows to send messages for various destinations over the same connection and to dispatch them on the receiver side.

- Flexibility in terms of payload
  The message format to use shall be able to carry all kinds of payload. This includes integers (signed and unsigned), strings, byte buffers and arrays.

  Even nested messages shall be possible. This means that a message of type A can contain a message of type B that itself contains two messages of type C and so on.

- Flexibility in terms of transport method
  The message format shall be used for different kinds of transport. For example, such messages will be sent over TCP connections or put as payload into existing routing messages.

- Ease of use
  The message objects shall offer the functionality of parsing themselves from a string and serialising themselves into a string. This reduces the complexity in message handling significantly.

To fulfill all the requirements a Type-Length-Value (TLV) message format is used. Basically, a TLV message looks as follows:

Where Type denotes a 32-bit integer value which uniquely specifies of which type the message is, e.g. Service Look-up. Length contains the length of the whole message (Type field + Length field + Value field) as 32-bit integer. The value part may contain a base type or again a TLV message.

The example in Figure 3.3 shows a message that itself contains a message that represents a string of characters. The outer message has type 27 and a length of 21 bytes. The length is calculated as follows: 4 bytes for the type field, 4 bytes for the length field and, finally, the length of the value (13 bytes). The inner message has type 11 and a length of 13 bytes. The actual content of the inner message is the string “ABCDE”.

### 3.4 Message Types

After having specified the message format. It is necessary to specify the message types for the information exchange between processes. For the communication between the application layer and the message dispatcher the message types listed in table 3.1 are used. For a more detailed description of message types and the contents of each message type see appendix D.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD_PUBLISH</td>
<td>Sent by the application to the message dispatcher to publish a service.</td>
</tr>
<tr>
<td>MD_PUBLISH_RET</td>
<td>Used to answer a previously received MD_PUBLISH message.</td>
</tr>
<tr>
<td>MD_UNPUBLISH</td>
<td>Revokes a service publish.</td>
</tr>
<tr>
<td>MD_UNPUBLISH_RET</td>
<td>Used to answer a previously received MD_UNPUBLISH message.</td>
</tr>
<tr>
<td>MD_LOOKUP</td>
<td>Issues a service look-up request for a specific service.</td>
</tr>
<tr>
<td>MD_LOOKUP_RET</td>
<td>Used to return the potentially found service requested by an earlier MD_LOOKUP message.</td>
</tr>
<tr>
<td>MD_REGISTER</td>
<td>Has to be sent as first one by the application layer to register as handler for a service class.</td>
</tr>
<tr>
<td>MD_REGISTER_RET</td>
<td>Answers a previously received message of type MD_REGISTER.</td>
</tr>
<tr>
<td>MD_SET_STRATEGY</td>
<td>Using this message type the application layer can choose the strategy to use if more than one is offered, e.g. global knowledge or zero knowledge.</td>
</tr>
<tr>
<td>MD_SET_STRATEGY_RET</td>
<td>Reply for a previously received MD_SET_STRATEGY message.</td>
</tr>
</tbody>
</table>

Table 3.1: List of all message types.

### 3.5 Data Types

It is necessary to define some payload for the message types. For the service discovery architecture it is needed to exchange the types of data listed in table 3.2 as payload of the previously defined message types. For a more detailed description of data types see appendix C.

### 3.6 Transport Layer

The main problem to be solved in the transport layer is to get routing messages from the network and provide them to upper layers. Furthermore, the transport layer needs functionality to insert application specific data into routing messages and get such data out of routing messages if there is any.

The easiest way to achieve this would be to directly interface with the routing daemon using an API offered by the daemon. But this this is just in rare cases possible because most routing daemons do not offer a corresponding interface.
### 3.7 Message Dispatcher

On one hand, the message dispatcher gets messages from the transport layer and processes them. On the other hand, it gets messages to process from the application layer. The whole service discovery strategy, i.e., what to do at the arrival of which type of message, must be encapsulated. This allows the quick implementation of new strategies like a global knowledge strategy instead of a query-based strategy.

The message dispatcher layer is divided into two rather independent parts plus two parts that act as a kind of glue layer (see Figure 3.4):

- **Socket Listener**

  The socket listener part is responsible for the communication with the application. It gets service look-up requests and service publish requests from the applications and handles them or passes them down to the lower part. The socket listener is completely independent of the strategy used and even more independent of the transport layer.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceClass</td>
<td>An unsigned 32-bit integer to indicate the class of a service, as explained in section 2.4.1.</td>
</tr>
<tr>
<td>ServiceID</td>
<td>A string of characters to identify the actual service (see section 2.4.1).</td>
</tr>
<tr>
<td>ServicePayload</td>
<td>A byte buffer containing the payload of the service.</td>
</tr>
<tr>
<td>ServiceStrategy</td>
<td>An integer identifying a strategy.</td>
</tr>
<tr>
<td>ServiceStrategiesSupported</td>
<td>An array of integers containing all available strategies.</td>
</tr>
<tr>
<td>ServiceTimerMode</td>
<td>An integer describing a mode for service timeouts.</td>
</tr>
<tr>
<td>ServiceLookupMode</td>
<td>An integer describing the mode of a service look-up.</td>
</tr>
</tbody>
</table>

Table 3.2: List of all data types.

Another possibility would be to patch the routing daemon and add such an interface. This is not the best solution because of the constraint that the transport layer shall be as independent as possible from the routing protocol and/or daemon. The idea behind this is that the whole system can be easily adapted to other routing protocols and routing message types. Therefore, a direct interfacing with the routing daemon has to be avoided.

The most independent way to get such packets is to intercept them on their way from the network up the IP stack to the routing daemon and, in the opposite direction, on their way down the IP stack from the routing daemon to the network.

When a routing message arrives, it is first intercepted by the transport layer, passed up to the upper layers of DSD and finally, when it is returned by the upper layers, forwarded to the routing daemon.

The same holds analogously for the opposite direction. This makes it possible to change packets in a way that the routing daemon is not even aware of that.
3.7 Message Dispatcher

- Transport Handler
  
The transport handler lies between the transport layer and the socket listener. It gets the messages intercepted from the IP stack by the transport layer and processes them. On the other side, it gets work to do from the socket listener such as service look-up requests or service publish requests.

- Service Handler Store
  
The service handler store is used to store information about registered service handlers.

- Service Store
  
  This class is used to store all information about published services and service look-ups that are requested or ongoing. It is also the way of communication between socket listener and transport handler. If a new entry is made in this store, the transport handler gets notified so that it can process this new entry.

3.7.1 Socket Listener

The communication between socket listener and application layer (also called service handler) is done using sockets. It uses the messages specified in section 3.4.

An application that wants to use distributed service discovery has to register itself in a first step at the socket listener. After a successful registration it can publish and look up services.

Registration of a Service Handler

The registration consists of the exchange of several messages. The following steps are done (see Figure 3.5):

1. The application sends a MD_REGISTER message to the socket listener.
This message contains the service class for which the application wants to register.

2. The socket listener checks whether a handler for this service class is already registered.

   If there is no service handler for the corresponding class, a new service handler entry is created and stored in the service handler store. Then a \texttt{MD\_REGISTER\_RET} message is sent to the application containing the available strategies for service discovery and the preferred strategy.

   If there is already a service handler registered for the corresponding service class, a \texttt{MD\_REGISTER\_RET} message is sent containing a negative return value. The registration failed.

3. In case of successful registration, the application chooses a strategy and sends a \texttt{MD\_SET\_STRATEGY} message containing the chosen service strategy and \texttt{Service\_Timer\_Mode} to the socket listener.

4. The socket listener sets the strategy and \texttt{Service\_Timer\_Mode} entries for the corresponding service handler to the values chosen and replies with a \texttt{MD\_SET\_STRATEGY\_RET} message.

### Publishing Services

To publish a service the service handler sends a \texttt{MD\_PUBLISH} message to the socket listener, see Figure 3.6. The message contains service class, service identifier and service payload of the service to be published. The socket listener creates an new entry for this service in the store and marks it as newly published. Then it notifies the transport handler about the newly received publish request.

Basically a service that was once published remains in this state until it is explicitly unpublished by the responsible service handler. To do so, a \texttt{MD\_UN\_PUBLISH} message is sent to the socket listener.
As an alternative to this behaviour a timeout can be assigned globally to all publish messages of a service class. If this option is used, a service publish needs to be renewed periodically. If this is not done, the entry for the published service is deleted and the service is no longer discoverable.

Looking up Services

If a service handler sends a MD_LOOKUP message (see Figure 3.7), the socket listener checks if the requested service is already stored. If so, it answers with a MD_LOOKUP_RET message containing the requested service. If the service is not yet known, it extracts the service class and service identifier out of the MD_LOOKUP message, constructs a service entry with an empty payload, marks it as service look-up requested and stores it. Afterwards it notifies the transport handler about the new service look-up request.

A service look-up can take rather a long time. It is not even sure whether a service look-up that is sent over the network gets answered at all. This depends on the strategy used and on the availability of the service. Therefore, a timeout needs to be assigned to a service look-up request after which the request is deleted.

While a service look-up is in progress no new service look-up request for the same service will be issued to the lower layer. This default behaviour can be overridden setting the appropriate flag in the MD_LOOKUP message.
3.7.2 Transport Handler

The transport handler processes requests from two sides, from the transport layer and from the socket listener. The transport handler needs to be specific to the strategy as well as to the routing protocol used to transport the service information. But these dependencies need to be encapsulated and hidden from the main part of the transport handler.

Requests received from the Socket Listener

The transport handler gets notified by the socket listener in case of one of the following two events: a service was newly published by a service handler or a service look-up request to an unknown service was issued.

In both cases, the transport handler finds out what strategy is to be used for the service class of the service in question and calls the corresponding handler function.

This handler function decides whether a message has to be sent using the transport layer or not. There are only two types of messages sent to the network: MDLOOKUP in case of a service look-up needs to be sent or MD_PUBLISH in case a service look-up needs to be answered or service information needs to be sent proactively to the network.

Messages received from the Transport Layer

If a message is received from the transport layer, the transport handler first gets the payload of the message and parses it into a TLV message. According to the type and the service class of the TLV message (MD_PUBLISH or MD_LOOKUP) the corresponding strategy is looked up and the message handling functions are called for this strategy.

3.8 Service Handler

On application level a service handler needs to be provided that communicates with the socket listener. These service handlers can publish and look up services of the specific service class they register to. There can only be registered one service handler of a service class at a node. If more than one service handler is required for the same service class, this problem has to be solved on application level.

The service handler needs to register itself at the DSD system. After having successfully registered, the service handler can send arbitrarily MD_PUBLISH, MD_UNPUBLISH and MD_LOOKUP messages. See section 3.7.1.

3.9 Design of the SIP Service Handler (SIPHoc)

For the goal of this thesis, a working SIP infrastructure for MANETs, a service handler has to be written that provides the SIP URI to IP address mapping.

The constraint is to change as less as possible on existing SIP components. As discussed in section 2.5, some changes in an existing SIP proxy server are sufficient. What changes exactly are necessary is fully depending on the implementation of the SIP proxy server and therefore discussed in chapter 4.
The SIP user agent has not to be changed at all. It is just a configuration issue there.
Chapter 4

Implementation

This chapter describes implementation details of the DSD system and of SIPHoc. The description is held on class level and, where interesting, on function level. For further implementation details refer to the separate API documentation or to the source code.

4.1 Development Environment

The hardware as well as the software used for the implementation of this project are described in this section.

4.1.1 Hardware

As hardware platform for development and testing a pool of five Intel-based notebooks and two Intel-based workstations could be used. Network communication between the notebooks was done using built-in 802.11b wireless LAN. In addition, a wired LAN was used between the notebooks and the workstations for administrative purposes and to deploy the software, see Figure 4.1.

For development and testing a true multihop wireless network between the notebooks was necessary. The notebooks needed to be setup in order that a

Figure 4.1: Network setup of the development and testing computers.
4.1 Development Environment

Communication impossible Communication possible

Figure 4.2: Linear multihop network setup of the notebooks.

communication from the first one to the last one goes over four hops. This is
difficult to achieve physically; therefore, it was done logically. The notebooks
had a firewall setup that allowed packets to be received from exactly one node
and packets to be sent to exactly one node, see Figure 4.2.

4.1.2 Software

Operating System

The requirements for an operating system for this project are:

• Good low-level interfaces
  
  Since we need to do packet intercepting and manipulation of network
packets, we need an operating system that allows us to do these things in
a convenient way.

• Support for MANETs
  
  At least the implementation of a MANET routing protocol is needed.

• Support for wireless LAN interfaces

• Firewall / Portfilter functionality
  
  To realise the multihop setup for the notebooks a working, convenient
firewall is necessary.

• Availability of open source SIP components
  
  Since it is intended to change an existing SIP proxy server to use the
DSD system, a working open source SIP proxy server is required.

• Availability of the source code of as many components as possible
  
  In case of implementation problems, the availability of the source code
of all components may reduce the time for bug-fixing significantly.

All of these requirements are met by GNU/Linux. So we decided to use
GNU/Linux as operating system.

Development Tools

We used mainly GCC and the GNU tool-chain as development tools.
4.2 Common Utility Classes

4.1.3 Programming Languages
For the implementation of this project two programming languages were used. The DSD system is written in C++ whereas the service handler for SIP is written in Java.

C++
C++ was mainly used as implementation language because of its convenient low-level programming capabilities. Another reason to use C++ was the dependency on a library that offers a C API.

Java
Java was required as additional programming language because of the SIP proxy server to be used to integrate the service handler is implemented in Java. To make it possible to write the service handler in Java the whole IPC part of the project needed to be ported to Java.

4.2 Common Utility Classes
This section describes some of the helper classes used in different parts of the DSD system and SIPHoc.

4.2.1 Threading
The use of multi-threading results in the need of two additional features. The first one is mutual exclusion. To provide this the class Mutex was implemented. The second one is that a thread may want to wait for the completion of an operation done by another thread; or it may just wait for an event like the reception of a message. This functionality is provided by the Notifier class.

Thread
Multi-threading in C/C++ involves the use of several POSIX functions. Therefore, it is rather uncomfortable and error-prone. For this reason, the abstract base class Thread was developed. It offers a Java-like way to work with threads.

To use multi-threading, a derived class has to be programmed that implements the run() function. The thread is then started calling the start() function and can be joined using the join() function.

Mutex
This class is actually just a wrapper around the POSIX mutual exclusion functions. It makes work easier because it does all the initialisation stuff automatically. Besides this, it provides a clean object-oriented way of handling mutual exclusion.
4.2 Common Utility Classes

TLVObject class with derived classes.

Notifier

The implementation of the Notifier class uses POSIX condition variables to allow one or more threads to wait until they get notified by another thread. For this purpose, the functions wait(), notify() and notifyAll() are used.

4.2.2 TLV Classes

The TLV message format is represented by a class called TLVObject. TLVObject serves as base class for the implementation of all message types, like TLVinteger or TLVBuffer, see Figure 4.3.

TLVObject

One of the most important utility classes is TLVObject. The core functions of this class are writeTLVBinary() and readTLVBinary(). These functions allow the object to serialise and de-serialise itself to/from a byte buffer.

Both functions work similarly. The class contains a vector of TLVObjects. In case of a nested TLV message type all inner message types are put into that vector. What the functions do is they iterate over the vector and call themselves recursively for all elements in the vector.

The recursion is stopped in the derived classes of TLVObject that do not contain derived classes of TLVObject themselves, e.g. TLVinteger. These classes overwrite the functions and do the actual work. This means that they serialise themselves into the byte buffer provided or read themselves from the byte buffer.

To work with the TLVObject class is easy. For each TLV message type a derived class of TLVObject is made. Two cases can be distinguished: the more complicated case is the one where the derived class is a kind of base data type, i.e. contains no other sub-classes of TLVObject as members. In this case, at least the readTLVBinary(), writeTLVBinary() and the getTLVText() functions need to be overridden by a function appropriate for the contents of the base data type. An example for this can be found in the source code of TLVinteger.
The less complicated case is a derived class that contains just some sub-classes of \textit{TLVObject}. The only thing that needs to be done is basically to call the function \textit{addElement()} for each of the sub-objects. An example can be found in the source code of \textit{TLVipv4}.

\textbf{TLVListenerTCP}

Receiving and sending TLV messages over a TCP connection in server mode is the core functionality of the class \textit{TLVListenerTCP}.

The class is implemented as thread. Firstly, a new TCP socket is opened and put into listen mode, see Figure 4.4. Secondly, the thread waits for a new connection to this socket using \textit{select()}. If a new connection is initiated, the thread assigns a client identifier to this connection and puts it together with a \textit{Mutex} object into an internal data structure. The \textit{Mutex} is used to make the class thread safe. This is necessary because the concurrent sending and receiving of messages on one TCP socket can cause problems. Afterwards, it waits again for a new connection or for activity on an existing connection. This can be incoming data or the shutdown of the connection.

In case of incoming data, the function \textit{processIncomingMessage()} is called which reads the data from the socket and puts it into a \textit{SyncedFifoQueue} together with the client identifier of the connection on which the message came in. Finally, all listeners interested in receiving messages are notified. These listeners then call the \textit{receiveTLVObject()} function to get the message together with its client identifier.

The client identifier is normally used as argument to the \textit{sendTLVObject()} function when sending a reply to the message received. This function can be called as a blocking function from outside of the thread. It uses mutual exclusion to prevent concurrent sending and receiving on the same socket.

A special feature of \textit{TLVListenerTCP} is the handling of closed connections. For the user of this class it might be necessary to get notified if a connection gets closed. If the thread detects that a connection was closed, it puts the client identifier of this connection into an internal datastructure and notifies all listeners interested in that kind of events. The listeners can then get the identifier of the closed connection.

The reason not to directly use file descriptors as identifiers instead of the special client identifiers is that the file descriptors get reused very quickly. Hence, if a connection gets closed, the file descriptor might be in use again for a new connection before the listener interested in close events comes into action.

\textbf{TLVTCPSocket}

This is the counter part to the class \textit{TLVListenerTCP} and allows to open a TCP connection to a server for sending and receiving TLV messages over it. This class is not implemented as a thread and messages are sent or received by the thread that uses this object. There is one blocking function \textit{sendObject()} to serialise \textit{TLVObject} and send it to the server.

TLV objects sent from the server to the client can be received either by \textit{receiveObjectBlocking()} or \textit{receiveObjectNonBlocking()}. As the name suggests, the first one waits until a message is received before returning. The latter is
4.2 Common Utility Classes

non-blocking and returns 0 if no message is there to receive. To check whether the socket is ready to receive, the function `isAvailable()` can be used.

4.2.3 Miscellaneous

This is a selection of some interesting utility functions that are not as important as the ones mentioned above.

TimerThread / Timer

The `TimerThread` offers the possibility to schedule actions that shall be done at some point in time in the future. The implementation uses signals to provide this functionality. It is important that the `TimerThread` is the only thread that uses these specific signals. The reason for this is the non-POSIX-conform behaviour of the GNU/Linux kernels used, for details please refer to the source code.
4.2 Common Utility Classes

The TimerThread is implemented conforming to the singleton pattern. As a specialization it uses reference counting to decide whether it is needed any longer or not. Every thread that wants to use timers needs to call the static function `startTimerThread()`. After this call it can use timers. If a thread no longer needs timers, it calls the `stopTimerThread()` function. If no thread uses timers anymore, the TimerThread is stopped.

To use the TimerThread the abstract base class Timer needs to be subclassed. An example for this can be found in the class MyTimer which is a test class.

SyncedFifoQueue

As the name implies this class provides a synchronised first-in-first-out queue. Synchronised means that the queue is thread safe. The implementation uses a STL deque as internal data structure. As a special feature the SyncedFifoQueue offers the possibility to register a notifier. This notifier is then used to notify all interested threads when an entry is put into the queue or got from the queue.

The SyncedFifoQueue is a template class. Therefore, it supports all data types that are also supported by the STL deque.

InterfaceInfo

Sometimes it is necessary to find out which IP addresses are assigned to the network interfaces. This is done in the InterfaceInfo class. It provides a convenient way to get information about interfaces and IP addresses.

The InterfaceInfo class provides functions to get interface names and the corresponding IP addresses for these interfaces. It is also possible to get just the names of the wireless interfaces using the function `getWirelessInterfaces()`.

The core function of this class is taken from [26].

NumberPool

This class solves the problem of wrap-around when using integers as identifiers. Basically, it provides the functions `getNext()` and `release()`. While the first one gets a new unused value, the second one releases a used value.

Internally, a STL hash_set is used to account for the used numbers.

BaseConfigurator

The BaseConfigurator class is used for the configuration of the whole software system. It is an abstract base class that provides the features of reading arguments from command line as well as from a configuration file. For reading command line arguments the `getopt_long()` function is used which is provided by most libc.

To read the config file a parser for Ini-syntax was written. For an EBNF description of the configuration file syntax see listing B.1. For an example configuration file see listing 4.1.

The name of the configuration file is specified as command line argument `-c`. Command line arguments are processed with a higher priority than the values in the configuration file. It is possible to have arguments that can be command line only, arguments that can only be specified in the configuration file and
4.2 Common Utility Classes

arguments that are specified in both, the command line and the configuration file.

```plaintext
[AODVRoutingHandler]
IPPoolStartIP = 10.0.0.210
IPPoolEndIP = 10.0.0.254
FreeDelay = 10
TriggerPort = 5555
AODVNetDiameter = 35

[IPC]
Port = 8480

[IPQ]
PollInterval = 50000

[Timeouts]
PublishTimeout = 10
RemotePublishTimeout = 15
LookupRequestTimeout = 15
LookupProgressTimeout = 15
```

Listing 4.1: Example of a configuration file.

To use this class a derived class has to be created that implements the `init()` function and provides all the information about command line and or configuration file arguments. This is done by building a vector of structs containing this information and passing them to the `setProperties()` function. Default values are set at this place, too.

The derived class should be implemented as singleton in order to be accessed from everywhere. For an example see the `TestConfigurator` class.

Logger

Debugging a distributed application is hard enough, a consolidated log file of all threads within a process can help a lot. Each thread could just print its log messages to a file or to `stdout`, but it happens often that two threads are writing their log messages just at the same time and they get interleaved. To solve this problem, a lightweight logging facility was implemented. Locks are used to separate the output of each thread.

There are 4 different log levels as described in table 4.1. A special case is log level `critical`, which terminates the program by calling the function `exit(1)` after printing the log messages. All log messages are formatted as follows:

```
<timestamp> <thread id> <log level> [ <filename>:<line number>]: <log message>
```

The filename and line number indicate in which source file at what position the log statement was executed.

An example log message looks as follows:

```
```

1 Time format: YYYY-MM-DD_hh:mm:ss:s
4.3 Distributed Service Discovery System

We decided earlier to implement a distributed query-based approach using a reactive routing protocol. Now the question arises what routing protocol to use.

The most-widely used reactive routing protocol is AODV. Since the AODV implementation by the University of Uppsala is well tested and working flawlessly, we chose to implement distributed service discovery for the AODV routing protocol using the AODV-UU implementation for testing.

The following sections describe the implementation of the DSD system in detail.

4.3.1 Generic Part of the Transport Layer

The transport layer is subdivided into several classes and interfaces:

- **IPQThread**

  This is an interface for the class responsible for the actual interception of packets. It shall be implemented as an own thread in order that there is as little delay as possible when intercepting a packet. The interface hides the actual mechanism used for intercepting the routing packets.

- **BaseTransportMessage**

  BaseTransportMessage is an abstract base class. Every time the IPQThread implementation intercepts a packet, it is wrapped into a BaseTransportMessage instance and passed over to the upper layer.

  This class is fully independent of the actual contents of the message. It just provides a byte buffer where the raw message is stored. Specific implementations for routing protocols have to provide a protocol specific sub-class. This is the place where the routing message format specific functions, how to put payload data into messages and how to get it from messages, are put.

- **AbstractRoutingHandler**

  This is the interface that must be implemented for specific routing protocols, see Figure 4.6. It is used by the upper layer to manipulate instances of the protocol specific implementation of the BaseTransportMessage class. It offers functionality like sending messages or answering to received messages.

Using this class layout, the dependency on a specific routing protocol is encapsulated into two classes: an implementation of AbstractRoutingHandler.
4.3 Distributed Service Discovery System

and a derived class of `BaseTransportMessage`. Hence, these two classes are the only thing that has to be reimplemented to adapt the DSD to a new routing protocol.

The way of intercepting the routing messages is hidden behind the interface `IIPQThread`. Therefore, it is easy to change the packet intercepting if necessary.

**Intercepting Network Packets using LIBIPQ**

One of the main problems to be solved in the implementation of the transport layer is to intercept packets while they are travelling the IP stack up from the network or down to the network.

Normally, this is done under GNU/Linux in a kernel module using `netfilter` [27] functionality. But `netfilter` also provides a much more comfortable way to intercept and mangle packets called `LIBIPQ`.

`LIBIPQ` provides easy to use functions to intercept packets as a user-space application. To determine what kind of packets are to be intercepted one or more `iptables` rules are used. If for example all AODV packets that are incoming to the node shall be intercepted, the `iptables` rule shown in listing 4.2 needs to be set before the application is started.

```
root@node1> iptables -A INPUT -p udp --dport 654 -j QUEUE
```

Listing 4.2: Setting up an `iptables` rule to intercept AODV routing packets.
This sends all incoming UDP packets with destination port 654 to a queue. The packets can then be read from this queue using the \textit{LIBIPQ} function \texttt{ipq\_read()}. Since 654 is the port assigned to the AODV protocol, this intercepts all AODV messages received.

As a prerequisite the following two kernel modules need to be loaded: \texttt{ip\_queue}, \texttt{iptable\_filter}

After having received the message it can be processed and mangled. When the application is done, the message is put back to the IP stack using the function \texttt{ipq\_set\_verdict()}, see Figure 4.7. Packets can also be dropped instead of sending them back to the IP stack.

For a more detailed example on the usage of \textit{LIBIPQ} refer to the \texttt{man}-page or to [28].

\textbf{BaseTransportMessage Class}

The \textit{BaseTransportMessage} class holds all the information that is not specific to the routing protocol used. This means that there is mainly a byte buffer containing the raw message received from the network. This byte buffer is
called messagebuffer.

There are also some utility functions like `calculateUDPIPChecksum()` and `doesHaveChanged()`. These functions serve to recalculate the checksum of the packet after having changed the content. This is necessary because we operate with raw network messages and therefore, there is no automatic calculation of checksums nor any validity check before sending the message back to the network layer. To prevent unnecessary checksum calculations, every function that changes the byte buffer must set the `hasChanged` flag.

The most important points to mention in this class are the two abstract functions `setData()` and `getData()`. This functions operate on the message buffer and are called to get the actual message payload from the routing protocol specific place or to put it there. The functions must be implemented in a derived class that is specific for the routing protocol.

**AbstractRoutingHandler Interface**

This interface is used to hide the routing protocol specific functions for sending messages. The interface provides functions that a service discovery strategy implementation may use, see Figure 4.8. An implementation of a routing handler has to provide the following functions:

- **ignoreMessage()**
  
  This function is called if a received service discovery message shall be ignored. Correspondingly, no further actions shall be taken with the message. The underlying routing message is normally delivered to the routing daemon.

- **acceptMessage()**
  
  This means that the service discovery message was bound for this node. It shall not be sent away anymore, and the transporting routing message shall be delivered to the application as if it were sent to this very node, if this is appropriate for the routing protocol.

- **acceptAndAnswerMessage()**
  
  This function is basically used like `acceptMessage()`. The difference is that an answer message shall be triggered carrying the payload provided as argument.

- **sendMessage()**
  
  The buffer provided as argument shall be sent out to all nodes using routing messages as long as `sendMessageStop()` is called.

- **sendMessageStop()**
  
  This stops the sending of a message previously started with a call to `sendMessage()`.

- **processTransportMessage()**
  
  A call to this function must convert the `BaseTransportMessage` provided as argument into an instance of the routing protocol specific transport message. The protocol specific routing message has to be returned cast to a `BaseTransportMessage`.
4.3 Distributed Service Discovery System

<<interface>>
AbstractRoutingHandler

+ignoreMessage(BaseTransportMessage*)
+acceptMessage(BaseTransportMessage*)
+acceptAndAnswerMessage(BaseTransportMessage*, u_int8_t*, u_int32_t)
+sendMessage(u_int8_t*, u_int32_t): u_int32_t
+sendMessageStop(u_int32_t id)
+processTransportMessage(BaseTransportMessage*) : BaseTransportMessage*
+processDownstreamMessage(BaseTransportMessage*)

Figure 4.8: AbstractRoutingHandler interface.

Figure 4.9: The main loop of IPQThread.

This is used in the transport handler to convert a BaseTransportMessage into a specific one.

- processDownStreamMessage()

Called by the transport handler every time a downstream message is provided by the IPQThread, i.e. a message from the routing daemon to the network.

IPQThread

The IPQThread is the lowest layer in the whole architecture. It uses the low-level LIBIPQ functions to intercept the routing messages. It offers non-blocking functions to the transport handler to send, drop or get messages. These functions work with queues. This means a call to sendMessage() just puts the message into the input queue of IPQThread while a call getMessage() gets the message from the output queue of IPQThread.

The core part of the class is the run-loop. As long as the IPQThread is active it processes this loop. First of all, it checks if there are some messages to be sent in its input queue, see Figure 4.9. If so, these messages are sent back to the IP stack using LIBIPQ functions. After this it tries to get the next intercepted message doing a call to the ipq_read().

A problem with LIBIPQ is that it is not thread safe, which means that it is not allowed to send and receive messages at the same time. That is the reason why in this loop the IPQThread needs to process the messages to be sent first and then needs to check for new messages. Otherwise two threads could be used, one for receiving and another one for sending asynchronously.
4.3 Distributed Service Discovery System

What is done in the run-loop is basically polling. First, the input queue gets polled then the LIBIPQ—as a result, we actually have a busy wait. More beautiful would be to use I/O-multiplexing but this is impossible because LIBIPQ does not offer this functionality so far. The only thing that can be done is to set a timeout for the ipq_read() function.

In the implementation, this timeout is set by default to 50ms. If the timeout is set too big, the routing messages get delayed too long and this can cause difficulties. In case of the AODV protocol, the routing daemon expects HELLO messages from his neighbours once about every second. If these messages are not received, a link break is supposed and the corresponding measures are taken. Therefore, it is really important to find a trade-off between busy wait and delaying messages for too long.

4.3.2 AODV-specific Part of the Transport Layer

The implementation of the AbstractRoutingHandler interface is done in the RoutingHandlerImplAODV class. This is actually the most complicated part of the whole transport layer of the DSD for the AODV protocol. It is necessary to understand the working principles of AODV and to take use of all details of the protocol. Since this is the core part it is necessary to explain first of all some of the working principles of AODV.

Basic Working Cycle of AODV

As mentioned before, AODV as a reactive routing protocol works basically using two kinds of messages—route requests (RREQ) and route replies (RREP). If a route to an unknown host is required, the AODV daemon issues a RREQ message containing the destination address searched for and some other stuff, see Figure 4.10. This RREQ message is broadcasted until it receives the destination. To optimise this process an expanding ring search is used. This means that the IP TTL of the RREQ is in a first attempt set to 2. If no answer is received after a timeout, the TTL is increased and the RREQ is resent. This goes on until an answer is received or the TTL reaches a defined maximum.

At the destination, a RREP message is issued and unicasted back to the originator of the RREQ. If an intermediary node already has a route to the destination, then this process stops already there and a RREQ is answered by this intermediary node with a corresponding RREP.

The intention is now to use this working cycle for service discovery messages. This means service look-ups shall be sent piggybacked with RREQ messages and the corresponding answers shall be sent as RREP messages.

In order to do so, we need first of all a possibility to trigger RREQ messages every time a service look-up message must be sent over the network. Furthermore, we need a possibility to answer the service look-up messages with a corresponding service publish message or a RREP message, respectively.

\[^2\text{Actually it could be done by abusing the file descriptor returned at the initialisation of LIBIPQ in the struct ipq\_handle. But this struct is described as opaque in the LIBIPQ man-page.}\]
### AODV Transport Message Class

This is a derived class from `BaseTransportMessage` that extends its base class with some AODV-specific functions, see Figure 4.5.

Of course, there are first of all the `getData()` and `setData()` functions. The implementation is based on the AODV Extensions specified in [19] section 9. These AODV extensions offer the possibility to add user specific data to routing messages, see Figure 4.11. Using the extensions is quite easy. An existing AODV packet just gets lengthened by an extension header and the actual extension payload. The extension header contains an extension type value and an the extension length both as 8-bit integers.

A problem with these extensions is that they have a fixed size limit of 255 bytes. This is too little to transport sensible service discovery messages. To address this issue we used two different extension types. 129 to indicate the start of the service discovery specific extensions and 130 for every following extension. If now a payload larger than 255 bytes is put into a message, it is split into buckets of 255 bytes, see Figure 4.12. The use of two different extension types makes it possible to attach more than one service discovery message to the same routing message.

One major problem during implementation was a hard coded message size boundary of about 800 bytes in the AODV-UU implementation. If larger messages are received the AODV daemon may crash. This could easily be fixed by setting the value for the maximum message size to the maximum size of a UDP message.

Because we did want to work with an unpatched out-of-the-box AODV dae-
mon, the maximum size of a service discovery payload was limited to 300 bytes in the implementation.

**Triggering RREQ Messages**

What we need to trigger a RREQ is an IP address that belongs to the subnet routed by AODV and to which no route is yet established. If then an application wants to send a packet to this host, an RREQ is issued automatically. An easy way to do this is to try to send an empty UDP message to this host. This is done in the `triggerMessage()` function.

Important is now to use an IP address to trigger messages to which a route is never established. This can be achieved by using an IP address that is reserved exclusively for this purpose. After triggering this RREQ message the expanding ring search comes into action and RREQ messages are sent to the network until the maximum TTL is reached. Consequently, the service look-up message can be sent with every one of these messages and therefore can also benefit from this optimisation.

Since a route to this reserved trigger address will never get established, the address can be reused for triggering a new RREQ message after the expanding ring search has ended. A major problem is now that this process takes some seconds. This would mean that a new service look-up could only be issued after the last one had finished. Consequently, no concurrent service look-ups would be possible.

To enable concurrent service look-ups a pool of reserved IP addresses must be used. For every call to `triggerMessage()` a new IP address can be used. An IP address that was used to trigger a message is marked as unavailable for further calls to `triggerMessage()`. After a timeout, when the expanding ring search is finished, the IP address is marked as available again and can be reused. The implementation of this scheme is rather complicated and involves
the following classes: `RoutingHandlerImplAODV`, `IPPool`, `IPTimer`, `NumberPool`, `AODVContextStore` and `AODVContext`.

**Sending Messages piggybacked to AODV Messages**

The process of sending a message is illustrated in Figure 4.13. First the function `sendMessage()` is called with a buffer containing the data to be sent as argument. The data is sent to the network until this is stopped by calling the `sendMessageStop()` function. To do so the `sendMessage()` function returns an integer as identifier that is afterwards passed to the `sendMessageStop()` function to tell what message shall no longer be sent. The management of the identifiers is done in the class `NumberPool`. This is essential because the identifiers must be unique even in case of a wrap-around.

After getting an identifier for the message to be sent, a new AODV RREQ is triggered to a new trigger IP address as described above. The RREQ carries this trigger address in its AODV destination address field. Now an instance of `AODVContext` is created and filled up with the buffer containing the data to

![Figure 4.13: The sendMessage() function.](image-url)
be sent and the AODV destination address. This *AODVContext* is put into the *AODVContextStore*. At the same time a link between the trigger IP address and the identifier of the message is made by entering them into map called *idIpMap*.

The actual sending of the message is then done in the function `processDownStreamMessage()`. This function is called every time a message is sent from the routing daemon down the IP stack. This includes of course the RREQ we triggered. The function does a look-up in the *AODVContextStore* and if a payload is found for this RREQ it is put into the message. This is also the place where the timeout for the trigger IP address is scheduled by setting an *IPTimer* using one of the *setFree()* functions in the class *IPPool*.

If full cycle of expanding ring search is done and the message still needs to be sent, a new cycle is started in the function `processDownStreamMessage()` by triggering a new RREQ message.

**Answering to Messages using AODV Messages**

Answer to a received service discovery message means using an AODV RREP as answer to a AODV RREQ. This can be achieved by calling the function `acceptAndAnswerMessage()`.

When a node receives an AODV RREQ message containing some service discovery information, this RREQ is addressed to a reserved IP address, not to the one the node uses. This means forwarding this RREQ unchanged to the routing daemon would not lead to a RREP message. We need to change the RREQ message in a manner that it is not distinguishable from a RREQ that is addressed to this node. Subsequently, the routing daemon will accept the RREQ and construct a RREP to answer. The service discovery information can then be transported using this RREP message.

The problem that is easy to solve is to address the RREQ to the local node. This is done by simply overwriting the AODV destination address with the address of the incoming interface of the local node.

The more difficult problem is that a RREQ contains a *Destination Sequence Number*. Since the RREQ is actually addressed to a reserved trigger IP address, the destination sequence number has not a valid value for the local node. This should actually not be a problem, since the AODV standard defines the *Unknown sequence number* flag for RREQ messages. Unfortunately, the AODV-UU implementation seems not to use this feature.

The solution we choose is to store the sequence number that is contained in every regularly sent AODV *HELLO* message. Now we can fill in an appropriate destination sequence number after having changed the destination address in a RREQ message.

Appropriate means that the destination sequence number is incremented by one before inserting it into the RREQ message. This makes the routing daemon increment its local sequence number counter also by one so that the resulting RREP contains the highest sequence number seen so far for this node. Finally, all intermediary nodes forward the RREPs because they represent the most recent routing information. This would not be the case for routing messages containing an older destination sequence number.

The handling of the actual message payload to be sent works like in case of sending a message. This means that a *AODVContext* is created containing the data to be sent, but, in contrast to sending messages, the IP destination
address is used to identify the downstream packet and not the AODV destination address. This works because RREPs are unicasted and not broadcasted to the receiver.

4.3.3 Message Dispatcher

To implement the parts of the message dispatcher mentioned in section 3.7 some helper classes were necessary. To provide the required flexibility in terms of service discovery strategies an interface called AbstractStrategy was created. The actual strategy used is hidden behind this interface. During the thesis, an implementation of this interface for the Zero Knowledge Strategy was made.

AbstractStrategy Interface

The AbstractStrategy interface provides functions to handle all events related to messages that occur during service discovery. It is intended that these event handling functions use the functionality provided by the AbstractRoutingHandler interface to send and manipulate messages. The following list gives an overview over all functions of this interface:

- processPublish()
  
  This function is called when a service publish message arrives from the network. This is e.g. the case as an answer to a previously sent service look-up message.

- processNewPublish()
  
  When a new local service is published, this function gets called.

- processLookup()
  
  The processLookup() function is called every time a service look-up message is received from the network.

- processLookupRequest()
  
  This function is responsible for handling service look-up requests issued by an application.

- handleTimerQueueEvent()
  
  As described earlier in certain circumstances a timeout for services is needed. This function gets called every time a service entry has a time out.

Zero Knowledge Strategy

The implementation of the Zero Knowledge strategy is done in the StrategyImplZeroKnowledge class. The following list explains what all the functions do:

- processPublish()
  
  If this function gets called, a service publish message was received from the network. The following things are done after checking whether a matching entry is in the store:
4.3 Distributed Service Discovery System

- If there is already a locally or remotely published service in the store:
  Just ignore the message.
- If there is no entry at all:
  Just ignore the message.
- If there is service look-up request in progress for this service:
  Then we received an answer to this service look-up request. In this case, two things need to be done. The first step includes stopping the sending of further service look-up messages for this service. Secondly one has to send an answer to the service handler responsible for this class. Stopping the sending of more messages is a bit tricky.
  A call to the `sendMessage()` function of the routing handler returns an identifier that needs to be passed to the `sendMessageStop()` function. This means after having called `sendMessage()` to send out a service look-up request this identifier needs to be saved somehow in order that after getting an answer it can be used to stop the sending of service look-up requests. This problem is solved using a map called `sidIdMap`. After a call to `sendMessage()`, the identifier is stored together with the service identifier of the lookuped service in this map.
  Hence, after having received an answer to the service look-up request we just need to extract the service identifier from this answer and do a service look-up in the map to get the argument needed for calling `sendMessageStop()`.

- processNewPublish()
  Because we implement a Zero Knowledge strategy we do not need to do anything in this case. The function is just empty.

- processLookup()
  If a service look-up message is received from the network the following things are done after checking whether a matching service was locally published:
  - If so: answer with a corresponding service publish message.
  - If not: just ignore the message.

- processLookupRequest()
  The processing of a service look-up request consists of the following steps:
  1. Check if the service to look up became available in the meantime.
     If so: send an answer to the corresponding service handler and we are done.
  2. Send a service look-up message to the network using the features offered by the `AbstractRoutingHandler` interface.

- handleTimerQueueEvent()
  Timers are used in the Zero Knowledge Strategy implementation to tell that an entry in the `ServiceStore` has to be removed. This means the
Service publish has expired or a service look-up was not successful and timed out. The thing to do is to just delete the corresponding entries in the service store and in case of a lookup request stop the sending of further messages.

**ServiceStore and ServiceHandlerStore Classes**

The *ServiceStore* class is an important part of the DSD architecture. It is the connector between the socket listener and the transport handler. Every time the socket listener puts something into the *ServiceStore* the transport handler gets notified and checks if work is there for it.

The implementation uses a straight-forward approach. As main datastructure an STL vector is used. There are of course more efficient ways of implementing this, that could be used in case of performance problems.

The *lookup()* function of this store takes a *Service* object as argument. All fields in this service object that are filled in are taken into account when searching for entries in the store. Fields that are empty match to all values. There are also some more advanced look-up functions that take just service flags as arguments. The implementation uses mutual exclusion to be thread safe.

All the things mentioned are also true for the *ServiceHandlerStore* class analogously.

**Transport Handler**

The transport handler consists of the *TransportHandlerThread* class. It is the caller of the functions defined by the *AbstractStrategy* interface. The *TransportHandlerThread* consists mainly of a loop in which it waits on a notifier to get some work to do. It gets notified to process three different kinds of work.

1. A new message was received from the network.
   - In this case the *TransportHandlerThread* gets notified by the *IIPQ-Thread* implementation. It gets the message and calls the appropriate processing functions of a *AbstractStrategy* implementation. After the completion of the function the message is returned to the *IIPQThread* implementation to send it back to the network or drop it, whatever the verdict on the message was.

2. A service look-up request or a new local service publish is entered into the *ServiceStore*.
   - The *TransportHandlerThread* gets notified and questions the *ServiceStore* for new entries. Then it calls the strategy’s processing functions.

3. The timer for a service entry expired.
   - To process an expired timer the *TransportHandlerThread* gets notified and calls the *handleTimerQueueEvent()* function of the strategy used.

Since strategies are selected on a service class basis—and therefore also on a service handler basis—the *TransportHandlerThread* checks for every message or service it processes to which service handler it belongs. As a result, it can find out which strategy to use for the processing of an event. This is done using the *checkForServiceHandler()* function.
4.4 SIP Proxy Server using DSD

Socket Listener
The socket listener is divided into two classes:

- SocketListenerThread
  
  The SocketListenerThread is responsible for offering a socket to the application layer and for receiving and sending messages. It is implemented as a separate thread to prevent delays in sending or receiving messages.

- MDWorkerThread
  
  This the worker thread that can be started by the SocketListenerThread to process messages.

Despite the functionality offered by the SocketListenerThread it is a very small class. It has a main loop that accepts incoming connections and receives messages over existing connections. For the processing of every message a new instance of MDWorkerThread is started. The reason for that is to keep the delay in receiving messages as short as possible. For the whole connection stuff an implementation of the interface ITLVListener, the TLVListenerTCP, is used. This class is explained in section 4.2.2.

In its main loop the SocketListenerThread waits first for getting notified by the ITLVListener implementation. The notification may have two reasons. The first one embodies the reception of a new message. Secondly, an existing connection was closed.

In the first case, the SocketListenerThread gets the message and passes it to a new instance of MDWorkerThread. The MDWorkerThread processes the message in a straight-forward manner.

In the second case, the closed connection has to be handled. Getting notified for a closed connection means that the service handler that opened this connection is no longer running. What remains to do is the deletion of all entries in internal datastructures that belong to this service handler. Since a service handler is responsible for a specific service class. All entries of this service class in the ServiceStore are deleted. In addition to that the entry in the Service-HandlerStore gets deleted, too. This brings the system into the same state as if the service handler was never registered.

4.4 SIP Proxy Server using DSD

The first step in integration SIP into the DSD system is to find an existing SIP proxy server that can be used to do this. The proxy server needs to be open source and extensible, i.e. not too complicated to understand and not too complex.

After the evaluation some available products like ser we choose the JAIN-SIP Proxy developed by the National Institute of Standards and Technology (NIST) [29].

The JAIN-SIP Proxy is an open-source implementation of a full-featured SIP proxy server based on the Java API specified in JSR 32 [30].

Because this proxy server is written in Java, all the parts of the DSD infrastructure that are related to the application level interface had to be ported
4.4 SIP Proxy Server using DSD

to Java. This includes the whole TLVObject hierarchy, the BufferingTLVTCPSocket and some utility classes.

4.4.1 Porting TLV to Java

The port of the TLV part from C++ to Java was done in a straight-forward manner. Special attention was necessary on the low-level parts of the code. Whereas in C++ memcpy could be used to write integers into a byte buffer or read them from a byte buffer this is a bit more complicated in Java. We implemented helper functions in the TLVUtil class to do this job using logical bit operations.

Because the target Java version was 1.4, which does not offer Generics, the template classes needed to be converted into several normal classes. Another problem is that Java does not support unsigned integers. Therefore, for communication from C++ to Java it is important not to send unsigned integers that are too large because the handling of unsigned integers that are greater than the value range of a Java integer is not implemented\(^3\).

The TCP communication part was implemented using the java.nio package. The sockets in this package work buffer oriented in contrary to the stream oriented sockets in the java.net package. This is helpful because a socket in Java then offers more or less the same functionality as a socket in C++.

4.4.2 Integrating JAIN-SIP Proxy to DSD

As expected after the detailed analysis in chapter 2.5 making the existing SIP proxy use DSD was not difficult. The first step was to locate the SIP location service part of the JAIN-SIP proxy. It can be found in the package gov.nist.sip.proxy.registrar. The responsible class is RegistrationsTable. All changes made are local to this package.

A registered SIP user agent is represented as Registration object. Such an object contains all information about the registered user like FromHeader, ToHeader, UserName, DisplayName, Key and ContactsList. The really important entries are Key and ContactsList. All other entries can be reconstructed sufficiently when these two fields are known. Sufficiently means that all mandatory fields can be filled in with reasonable, correct values. Optional parts may be omitted. An example for an Registration object produced after the registration of a user can be found in listing 4.3.

Actually an easy way to do service discovery is using the Registration objects as service payload. Java offers functionality to serialise and deserialize objects. The only thing that remains to do would be to call the corresponding functions. We tested this approach and came to the conclusion that it works.

The drawback of this approach is that it is highly application specific. This kind of service payload can just be used to do service discovery for SIP using the same JAIN-SIP proxy servers on all nodes. The better solution is to define an application independent service payload for SIP services. To do this we should concentrate on the core problem. This is the mapping of a SIP URI containing a symbolic host address to a SIP URI containing the resolved address of the corresponding node.

\[^3\]This could be done by representing a C++ u_int32_t integer as a Java long.
4.4 SIP Proxy Server using DSD

Listing 4.3: Contents of a registration object.

If we take a closer look at listing 4.3, we find out that the information needed can be found in the Key field and in the ContactHeaders field. The Key is the canonical form of the SIP URI contained in the ToHeader field as defined in the SIP standard. This fact makes it the logical choice as service identifier for service discovery, see listing 4.4. To save space in the service discovery messages and to make them as generic as possible the information in the contact field can be stripped down to the absolutely necessary. This is the SIP URI part of the contact field, see listing 4.5.

Listing 4.4: Service identifier for SIP service discovery.

Listing 4.5: Service payload for SIP service discovery.

Using these definitions the service discovery for SIP application can also be used for non-Java SIP proxies or components that can not handle serialised Java objects. Another important point is that it needs far less space in the service discovery messages.

After having defined the service identifier and the service payload to be used for SIP service discovery, the next step is to integrate DSD into the JAIN-SIP proxy. This RegistrationsTable class contains a Hashtable where all the registered user agents are stored represented as Registration objects. This Hashtable needs to be replaced by a class that acts as service handler for the DSD system. Such a class has to offer the same interface as the Hashtable class. To allow an easy switch from a classic hashtable to the DSD system an interface called IRegistrationStore was created. We wrote two classes that implement this interface HashtableRegistrationStore and ServiceHandlerRegistrationStore. Both classes are implemented as singleton to prevent multiple instantiations. The first one

---

The serialised Java object is too big for the message size boundary of the AODV-UU implementation mentioned in section 4.3.2.
is just a wrapper around the normal *Hashtable* class. The second one offers the access to the DSD system.

This means that if *put()* is called in the *ServiceHandlerRegistrationStore* class the corresponding registration is made available for DSD. If *get()* is called, a service look-up is issued to the DSD system. The actual communication is done in the separate class *ServiceHandlerCommunication*. This class contains just three functions. A call to *init()* opens a TCP connection to the DSD system and registers the proxy server as service handler for SIP services. *sendreceiveMessage()* sends a message like MD_LOOKUP or MD_PUBLISH to the DSD system and waits a given period of time for the answer. Finally, a call to *shutdown()* unregisters the proxy at the DSD system. Internally, the class *BufferingTLVTCPSocket* is used to do this communication. The speciality of this utility class is that it buffers received messages. This means that when calling *sendreceiveMessage()* the message to be sent and the type of the expected answer message are specified. If now a message is received that does not match the specified answer type, the function tries to receive another message.

A test of this setup showed that unexpectedly many calls to the DSD system were made by the JAIN-SIP proxy. The explanation for this is that because in the original implementation which used an *Hashtable* as data structure to store the registrations, operations on the store were virtually for free. This changes with the use of a DSD connection which involves socket communication. A simple call to *containsKey()* now involves a call to the lower layer and the exchange of TCP messages. In the worst case, where the key is not in the store, this triggers a service look-up message to be sent to the network.

To improve this behaviour the interface *IRegistrationStore* was extended with functions that have only local effects. For example, a call to *containsKeyLocal()* just checks in a local data structure whether a registration is present or not but does not send messages to the lower layer. This had the side effect that the *RegistrationsTable* class, which uses the *ServiceHandlerRegistrationStore* as data member, had also to offer some functions in a local version. The *Registrar* class is the place where the functions are finally called. All of these calls needed to be investigated and where appropriate replaced by calls to the local functions.

The result of this effort is that the interaction with the DSD system could be reduced to the necessary minimum.

The SIP service handler does not use the default timeouts offered by the DSD system. It manages timeouts itself. This enables the service handler to adapt the lifecycle of service discovery perfectly to the SIP lifecycle of registrations.

Consequently the service expiration needed to be addressed. This was done by setting up a Java *TimerTask* that deletes all SIP services received from remote nodes. To realise this the class *RemoteExpiresTask* was implemented and scheduled for execution at regular intervals.

---

5This is important since messages, especially service look-up messages, can be delayed a rather long time. Think of a scenario where a service look-up request times out, i.e. no answer is received. Then we issue a publish message. Now an answer to the actually timed out service look-up request is received before the answer to the publish message.
Chapter 5

Debugging and Testing

This chapter describes the techniques used for debugging and testing this project. The two most important test classes are explained in detail.

5.1 Unit Tests

We wrote unit tests for most of the classes. The tests are executed using the command `make check`.

In C++, we used the unit test framework `Cppunit`. The tests of the Java part use the well-known `Junit` framework.

5.2 Integration Testing

Doing integration tests of a distributed software system is rather difficult. But it is absolutely required to make sure that the system works as expected. Two test classes were written to do integration testing, `TransportHandlerThread_Test` and `SocketListener_Test`.

5.2.1 Transport Handler Test

This class tests the integration of all components from the `IPQThread` up to the middle of the message dispatcher, i.e. the `ServiceStore`. The test needs at least two nodes to work. On the first node it is started in `publish mode` while on the second node it is started in `look-up mode`.

- Publish mode

  What happens in `publish mode` is that some services to publish are manually entered into the `ServiceStore`. For the `TransportHandlerThread` it seems as if these entries were made by the `SocketListenerThread`. So if now a service look-up message is received for one of these services the message is answered by the `TransportHandlerThread` sending a corresponding service publish message.

- Look-up mode

  In the `look-up mode` the `ServiceStore` is filled up with service look-up requests to the same services that are published in `publish mode`. The
result is that the `TransportHandlerThread` sends out service look-up requests for these services that get answered by the node that is in publish mode.

When the test is finished, it is automatically checked whether the node in lookup mode received the correct answers to all lookup requests it sent out. This makes it easier to change something in the existing code, because running the test shows immediately whether all is still OK or not.

Before starting the test, the kernel modules for LIBIPQ must be loaded and the `iptables` rules must be set as explained in appendix A.4. The test is then started with the command `latest`. For specifying in which mode to run the test the environment variable `PATTERN` is used. The complete command lines are shown in listing 5.1. The tests need to be run with root privileges.

```bash
root@node1> modprobe iptable_filter
root@node1> modprobe ip_queue
root@node1> iptables -A INPUT -p udp --dport 654 -j QUEUE
root@node1> iptables -A OUTPUT -p udp --dport 654 -j QUEUE
root@node1> PATTERN=PUBLISH ./latest

root@node2> modprobe iptable_filter
root@node2> modprobe ip_queue
root@node2> iptables -A INPUT -p udp --dport 654 -j QUEUE
root@node2> iptables -A OUTPUT -p udp --dport 654 -j QUEUE
root@node2> PATTERN=LOOKUP ./latest
```

Listing 5.1: Commands to start the transport handler test.

### 5.2.2 Socket Listener Test

Complementary to the `Transport Handler Test`, this class tests the DSD system from the `TransportHandlerThread` up to the application level. Basically, the test simulates an application level service handler which registers to the DSD system, sends some service publish and service look-up messages and finally unregisters. In addition, the behaviour when using default timeouts is tested, too. This makes the test case rather longish and complicated.

The goal was to make it possible to run this test without root privileges. Therefore, we needed to find a way to do the test without using `IPQThread`. Consequently, the class `IPQThread_Simulator` was implemented which offers the same interface as the `IPQThread`.

The difference is that the simulator does not send any messages to the network nor does it get messages from the network. Instead of sending messages to the network, it puts them just into a data structure. Receiving a message from the network can be simulated by passing the message to the function `putMessageIntoOutQueue()`.

Besides the `IPQThread_Simulator`, a `RoutingHandlerImplAODV_Simulator` needed also to be implemented. This class just overwrites two functions of the original AODV routing handler. This is necessary because normally the routing handler triggers messages to be sent to the network and expects to receive them from the `IPQThread`.

If the test is finished it is automatically checked whether the messages sent to the `IPQThread_Simulator` were as expected.

To run the test just type the command `./sltest` on a console.
5.3 Memory Management

Memory management is an error-prone and difficult part in developing a C++ application. To make sure that there are no illegal or wrong memory accesses it is essential to use a memory corruption detection tool. We used Rational Purify to test for memory corruption. Another function of this tool, the memory leak detection, proved to be very useful, too.
Chapter 6

Conclusion and Future Work

This chapter analyses the results of this thesis and draws some conclusions. Finally, it gives suggestions for further work based on DSD and SIPHoc.

6.1 Summary

6.1.1 Distributed Service Discovery

We succeeded in designing and implementing a solution for Distributed Service Discovery in MANETs. The solution is designed to be very flexible, generic and efficient. It can easily be adapted to almost all routing protocols currently used in MANETs. Furthermore, different service discovery strategies can be plugged easily into the system.

The solution is not specific for a special kind of service, e.g. SIP service, but supports generic services. Consequently, the DSD solution found in this thesis can be used to do service discovery also for services like DNS.

Since the solution uses existing routing messages as transport method it is efficient in terms of message and routing overhead. Because the communication between the DSD system and its client works using well-specified messages sent over a TCP connection, it is possible to write the clients in any programming language that is able to do TCP communication. Corresponding message types and classes are offered for C++ and Java.

6.1.2 SIPHoc: SIP for MANETs

By extending an existing SIP proxy server in order to use DSD instead of a local store for SIP registrations, it was possible to provide SIP in MANETs without touching the SIP user agents. There is just a very small change in the configuration for a user agent needed which conforms fully to the SIP specification. Moreover, the changes in the SIP proxy server were rather small. This means that adapting other SIP proxy server implementations for the use in MANETS should be easy.
6.2 Limitations

A limitation in the implementation in this thesis is that a service discovery message cannot be longer than 300 bytes. This is due to the hard coded message size boundary in the AODV-UU implementation for which the system was tested.

6.3 Future Development

There are many options for a future development of this project. The most obvious is to use the DSD part of the system to solve other service discovery problems in MANETs, like DNS.

Another interesting topic is to enhance the DSD system with caching mechanisms.

Other options are to implement further strategies like a global knowledge strategy. Also an interesting work would be to adapt the system to other routing protocols than AODV.

One could also put work in making the system more user friendly and developing client libraries for other programming languages. This includes also the testing for more different setups as the one used for this project.

It would be very interesting to analyse the performance and scalability of the system doing real world tests and to compare the results with simulations.

Finally to get a VoIP solution that is comparable to the ones for infrastructure based networks a QoS service is needed for MANETs. This problem could be solved by the integration of the QoS signalling solution developed in [1] into the SIPHoc system.
Appendix A

How to use

This chapter describes how to use the DSD system and SIPHoc. The building process and the configuration are explained as well as the running of the system. Finally, an overview is given on how to extend DSD with new features.

A.1 Platform Requirements

The following software setup is necessary in order to build and run the project:

- GNU/Linux operating system with kernel version > 2.4.8 or 2.6.x
- GCC 3.3
- cppunit > 1.10
- iptables > 1.2.11 (including libipq)
- Java 1.4.x
- Junit
- log4j > 1.2
- Ant > 1.6
- AODV-UU 0.9
- a SIP user agent, preferably kphone

For testing, a standard installation of Debian GNU/Linux 3.1 (Sarge) was used, but the project should run with every GNU/Linux distribution that fulfils the requirements.

The following tools are needed to build from scratch:

- libtool 1.5
- autoconf 2.59
- automake 1.9
A.2 Building Process

To build the project a comfortable environment can be used based on the GNU tools autoconf, automake and libtool. If building from scratch, i.e. not using the provided tarball, as a first step the shell script autogen.sh needs to be run.

Then the build process can be started in the well-known way by calling configure and make, see listing A.1. An installation is not necessary.

```
user@node1> ./ autogen.sh
user@node1> ./ configure
user@node1> make
```

Listing A.1: Building the project.

If for some reasons the project shall be built without using the cppunit library and consequently without unit tests, configure can be run with the argument --disable-cppunit.

Cleanup can be done as shown in listing A.2.

```
user@node1> make dist-clean
user@node1> ./ autoclean.sh
```

Listing A.2: Cleaning up.

A.3 Configuration

The configuration is split up into three parts. The configuration of the DSD system, the configuration of the JAIN-SIP proxy and finally the configuration of the SIP user agent.

A.3.1 Configuration of the DSD System

The DSD system can be configured either from command line or using a configuration file. The DSD system configuration mechanism relies on the BaseConfigurator class described in section 4.2.3. Therefore, it uses Ini-syntax for its configuration file. The configuration file to be used is specified using the command line argument -c. An example configuration file can be found in section B. Table A.1 lists all supported configuration options and their default values.

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AODVRoutingHandler</td>
<td>IPPoolStartIP</td>
<td>-s</td>
<td>10.0.0.220</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See section 4.3.2</td>
</tr>
<tr>
<td>AODVRoutingHandler</td>
<td>IPPoolEndIP</td>
<td>-e</td>
<td>10.0.0.249</td>
</tr>
<tr>
<td>Last address of the IP pool explained above.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AODVRoutingHandler</td>
<td>FreeDelay</td>
<td>-f</td>
<td>10</td>
</tr>
<tr>
<td>Delay in seconds after which a IP address used to trigger RREQ message can be reused.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.3 Configuration

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AODVRoutingHandler</td>
<td>TriggerPort</td>
<td>-t</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AODVNetDiameter</td>
<td></td>
<td>-d</td>
<td>45</td>
</tr>
</tbody>
</table>

Port that is used to trigger messages.

Net diameter value to which the AODV daemon is configured to. This is the maximum TTL to send RREQ messages.

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC</td>
<td>Port</td>
<td>-p</td>
<td>8480</td>
</tr>
</tbody>
</table>

Port used on which the DSD system listens for requests.

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPQ</td>
<td>PollInterval</td>
<td>-i</td>
<td>50000</td>
</tr>
</tbody>
</table>

Indicates the timeout used when trying to receive intercepted packets in µs. See section 4.3.1 for details.

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeouts</td>
<td>PublishTimeout</td>
<td>-u</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>RemotePublishTimeout</td>
<td>-v</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>LookupRequestTimeout</td>
<td>-w</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>LookupProgressTimeout</td>
<td>-x</td>
<td>15</td>
</tr>
</tbody>
</table>

Timeout after which a locally published service gets invalid.¹

Same as above but for remotely published services.¹

<table>
<thead>
<tr>
<th>Section</th>
<th>Option</th>
<th>Cmd Line</th>
<th>Default</th>
</tr>
</thead>
</table>

Timeout after which a service look-up request is considered to have failed.

Timeout after which a service look-up in progress is considered to have failed.

Table A.1: Configuration options for the DSD system.

A.3.2 Configuration of the SIP Proxy Server

The SIP proxy server is located in the directory `source/siphoc/sipproxy`. It expects a configuration file in this directory that is named like follows: `configuration-X.xml`. X has to be replaced by the hostname of the system.

There are already configuration files for all nodes of the test setup. An example can be found in appendix B.

The documentation of all the sections can be found in the JAIN-SIP proxy server documentation. There are only two interesting options concerning SIP-Hoc. Both of them are found in the section `SIP_STACK`. The first one is `stack_IP_address` and the second one is `DOMAIN`.

**stack_IP_address**

This option specifies the IP address that is used to listen for SIP messages. It has to be set to the IP address assigned to the interface that is used for SIP communication.

¹ Only when service handler activates default timeouts.
A.4 Running and using

**DOMAIN**

The DOMAIN option specifies for which domains the SIP proxy server is responsible. There needs to be an entry for every domain name used in SIP URI of users that take part in the SIPHoc system.

**Other Sections**

All other options have to remain unchanged. Only the configuration listed in the example configuration files was tested. If another configuration option is changed than the ones listed above the system possibly will not work as expected.

**A.3.3 Configuration of the SIP User Agent**

SIPHoc was tested using kphone as user agent, but it should work with every other user agent as well. The important thing is that the IP address specified in the configuration file of the SIP proxy server is configured as outbound proxy in the SIP user agent.

Using other applications than kphone it might be necessary to configure the same IP address also as registrar or registration server.

**A.4 Running and using**

The first step to be done for running is to load the kernel modules needed by LIBIPQ and to set the iptables rules in order to intercept routing messages. This is done using the shell script aodviptablesup.sh or manually by entering the commands shown in listing A.3. This step requires root privileges.

```
root@node1 > modprobe iptable_filter
root@node1 > modprobe ip_queue
root@node1 > iptables -A INPUT -p udp --dport 654 -j QUEUE
root@node1 > iptables -A OUTPUT -p udp --dport 654 -j QUEUE
```

Listing A.3: Commands used to setup LIBIPQ.

The next step is to start the DSD system. This is done using the executable main. This is the place where the additional command line arguments described in table A.1 can be used and/or an configuration file can be specified using the argument -c, see example in listing A.4. The main binary is found in the directory source/siphoc/messagedispatcher/. The DSD system has to be run with root privileges.

```
root@node >./main -c ./configuration.conf
```

Listing A.4: Example for starting the DSD system specifying a configuration file.

Then the SIP proxy server needs to be run. In order to do this, an Ant target is defined called run-text-proxy, see listing A.5. The SIP proxy server is located in the directory source/siphoc/sipproxy/. Make sure to have created a configuration file as described in section A.3.
A.5 Developing new Service Handlers

Basically a service handler is an application that uses the interface specified in sections 3.7.1 and 3.8. Since this interface is not specified in terms of function calls, but using TCP messages, the implementation of a service handler is language independent.

A good point to start when developing a new service handler is to take a look at the existing code. An example of a service handler written in C++ can be found in the test class `SocketListener_Test`. This class does a full cycle of service discovery. First it registers as service handler at the DSD system, then it publishes and looks up some services and finally it unregisters again. For an example of a service handler implemented in Java, take a look at the class `ServiceHandlerRegistrationStore` and its helper classes in the SIP proxy server part.

A.6 Developing new Routing Handlers

To adapt the DSD system to a another routing protocol than AODV there are mainly two things to be done. First one is to implement the interface `AbstractRoutingHandler`. This can be done analogously to the class `RoutingHandlerImplAODV`. The second thing to do is to implement a derived class of `BaseTransportMessage` that works on the message type of the routing protocol used. Further explanations can be found in section 4.3.1.

Finally, of course, make sure to instantiate the new routing handler class in place of the old one in the file `main.cpp`.

A.7 Developing new Strategies

The main thing to do when developing a new strategy is to implement the interface `AbstractStrategy`. Have look at the class `StrategyImplZeroKnowledge` to get an idea how this is done.
To integrate the new strategy into the DSD system it is important to choose an identifier for the new strategy. The identifier is an unsigned integer. In the file main.cpp this identifier has to be entered in the data structure supported-Strategies. Finally, an instance of the new strategy has to be put together with the chosen identifier into the strategyMap like it is done for StrategyImplZero-Knowledge.

A service handler can then choose which strategy it wants to use sending a MD_SET_STRATEGY message.
Appendix B

Configuration Files

\[
\text{FILE} := \text{SECTION} \{ \text{SECTION} \}
\]
\[
\text{SECTION} := \{ \text{NEWLINE} \} \[ \text{SECTIONNAME} \] \} \text{NEWLINE} \{ \text{PROPERTY} \}
\]
\[
\text{SECTIONNAME} := \text{STRING}
\]
\[
\text{PROPERTY} := \text{TRIMSTRING} \[=\] \text{TRIMQUOTESTRING} \text{NEWLINE}
\]
\[
\text{TRIMSTRING} := \{ \text{WHITESPACE} \} \text{STRING} \{ \text{WHITESPACE} \}
\]
\[
\text{TRIMQUOTESTRING} := \{ \text{WHITESPACE} \} \text{QUOTESTRING} \{ \text{WHITESPACE} \}
\]
\[
\text{QUOTESTRING} := \text{""} \text{STRING} \text{""} | \text{STRING}
\]
\[
\text{STRING} := \{ \text{ALPHA} | \text{NUM} \} \text{WHITESPACE} \} \text{STRING}
\]
\[
\text{ALPHA} := \text{A} - \text{Z} | \text{a} - \text{z}
\]
\[
\text{NUM} := \text{0} - \text{9}
\]
\[
\text{WHITESPACE} := \text{'}
\]
\[
\text{NEWLINE} := \text{"\n"}
\]

Listing B.1: EBNF description of the configuration file format.
Listing B.2: Example of SIP proxy server configuration file.
Appendix C

Data Types

\[\text{TLVuint32} = \text{u_int32_t}\]
\[\text{TLVuint16} = \text{u_int16_t}\]
\[\text{TLVuint8} = \text{u_int8_t}\]
\[\text{TLVint32} = \text{int32_t}\]
\[\text{TLVint16} = \text{int16_t}\]
\[\text{TLVint8} = \text{int8_t}\]

The base data types holding signed or unsigned integers with a length of 8-bit, 16-bit or 32-bit.

\[\text{TLVBuffer} = \{\text{Byte}\}\]
\[\text{Byte} = \text{u_int8_t}\]

The \text{TLVBuffer} is used to store a byte buffer. The buffer has no fixed length; it is calculated from the total message length.

\[\text{TLVString} = \{\text{Character}\}\]
\[\text{Character} = \text{u_int8_t}\]

The \text{TLVString} contains a string of characters.

\[\text{TLVipv4} = \text{TLVuint32}\]
\[\text{TLVipv6} = \text{TLVBuffer}\]

These classes store IP addresses.
ServiceClass = TLVuint32

A ServiceClass is used to do a coarse grained classification of services, e.g. SIP, DNS, etc.

ServiceID = TLVString

The ServiceID identifies a service uniquely together with a ServiceClass.

ServicePayload = TLVBuffer

The information associated with a service is contained in a ServicePayload object. This is what we get back if we do a service look-up.

ServiceStatus = TLVuint32

The ServiceStatus data type is used as return value in IPC messages to tell the destination whether there was an error processing the last message received or not.

ServiceStrategy = TLVuint32

The ServiceStrategy contains the identifier of a service discovery strategy that can be used for DSD.

ServiceStrategiesSupported = TLVBuffer

This data type contains a list of identifiers for service discovery strategies.

ServiceLookupMode = TLVuint32

This class is used to tell the DSD system whether a service look-up request shall be processed even if another one for the same service is already in progress or not.

ServiceTimerMode = TLVuint32

The ServiceTimerMode is used to tell the DSD system whether to use default timeouts or not.
Appendix D

Message Types

**MD_REGISTER** = **ServiceClass**

The MD_REGISTER message is used by a service handler to register at the DSD system. Since registrations are carried out on a per service class basis, the ServiceClass to register for is included in the message.

**MD_REGISTER_RET** = **ServiceClass** **ServiceStatus** **ServiceStrategy** **ServiceStrategiesSupported**

The MD_REGISTER_RET message is used by the DSD system to answer a previously received MD_REGISTER message. It contains the information received with the MD_REGISTER message and a ServiceStatus that indicates whether the registration was successful or not. In addition to that the ServiceStrategy field contains the service strategy preferred by the DSD system and the ServiceStrategiesSupported field contains a list of all supported strategies.

**MD_UNREGISTER** = **ServiceClass**

Once registered at the DSD system, the MD_UNREGISTER message is used to remove the registration in order that another service handler can register for the ServiceClass used.

**MD_UNREGISTER_RET** = **ServiceClass** **ServiceStatus**

The MD_UNREGISTER_RET message is used as return message to a MD_UNREGISTER message. The ServiceStatus field tells the destination whether the de-registration was successful or not.

73
**MD_SET_STRATEGY** = ServiceClass | ServiceStrategy | ServiceTimerMode

This message is sent by the service handler after the registration at the DSD system. The ServiceStrategy field tells the DSD system which service discovery strategy to use while the ServiceTimerMode specifies whether default timeouts shall be used or not.

**MD_SET_STRATEGY_RET** = ServiceClass | ServiceStrategy | ServiceTimerMode | ServiceStatus

This is the return message to the MD_SET_STRATEGY message to send back a status code to the service handler.

**MD_PUBLISH** = ServiceClass | ServiceID | ServicePayload

This message type is used for the communication between the service handler and the DSD system as well as for the communication between the DSD instances on different nodes.

In the first case, the message is used to publish a service. It carries all necessary information like the ServiceClass, the ServiceID and the ServicePayload.

In the second case, it is sent to answer a look-up request that arrived at a DSD instance from another node.

**MD_PUBLISH_RET** = ServiceClass | ServiceID | ServicePayload | ServiceStatus

The MD_PUBLISH_RET message is used to tell the service handler whether a previously sent publish request was successful or not. This is indicated using the ServiceStatus field.

Contrary to the MD_PUBLISH message this message is not used for communication between the DSD instances running on the different nodes but just for communication between the DSD system and the service handler.

**MD_UNPUBLISH** = ServiceClass | ServiceID | ServicePayload

The MD_UNPUBLISH message is sent by the service handler to the DSD system and tells that a service that was previously published is no longer available.

This message type is just used for communication between the service handler and the DSD system.
The DSD system uses this message type to answer a MD_UNPUBLISH message received by a service handler.

\[
\text{MD\_UNPUBLISH\_RET} = \text{ServiceClass} \mid \text{ServiceID} \mid \text{ServicePayload} \mid \text{ServiceStatus}
\]

The MDLOOKUP message is used for the communication between service handler and DSD system as well as for communication between the instances of the DSD system on different nodes.

In the first case, it is sent when the service handler wants to discover a new service. The information needed is contained in the ServiceClass and ServiceID fields. The additional ServiceLookupMode field tells the DSD system that the service look-up shall be processed even if another service look-up for the same service is in progress.

In the second case, the MDLOOKUP message is sent if the DSD system has no locally published service to answer the service look-up request received from a service handler.

\[
\text{MD\_LOOKUP\_RET} = \text{ServiceClass} \mid \text{ServiceID} \mid \text{ServicePayload} \mid \text{ServiceStatus}
\]

The MDLOOKUP_RET message is used by the DSD system to answer a MDLOOKUP message received from a service handler. It contains the payload of the service searched for and a ServiceStatus to indicate whether the service look-up was successful or not.

Contrary to the MDLOOKUP message this message is not used for communication between the DSD instances running on the different nodes but just for communication between the DSD system and the service handler. A MDLOOKUP message sent by the DSD system to the network is answered by a MD_PUBLISH message.
List of Figures

1.1 Transition to infrastructureless telephone networks .............. 6
1.2 Overview over all actors of SIP. ............................. 6
1.3 SIP Register Request. ........................................ 8
1.4 Typical SIP session initiation. ............................... 9

2.1 Resolving a SIP URI. ........................................ 11
2.2 Example: Service Discovery. .................................. 12
2.3 Comparison of three approaches to distributed service discovery. 13
   (a) Centralised approach to replace central servers. ............ 13
   (b) Virtual Backbone approach for replacing centralised servers. 13
   (c) Fully distributed approach for replacing centralised servers. 13
2.4 Composition of a service descriptor. ........................ 16
2.5 SIP Proxy Server with subcomponents ........................ 17
2.6 Distributed SIP Location Service. ............................ 18
2.7 Distributed SIP architecture for MANETs. ...................... 19

3.1 Coarse grained partition of DSD. ............................. 24
3.2 Example for using message types. .............................. 25
3.3 Example TLV message. ......................................... 25
3.4 The components of the message dispatcher layer. ............... 28
3.5 Registration process at the socket listener. .................... 29
3.6 Service publish process at the socket listener. ............... 30
3.7 Service look-up process at the socket listener. ............... 30

4.1 Network setup of the development and testing computers. ....... 33
4.2 Linear multihop network setup of the notebooks. ............... 34
4.3 TLVObject class with derived classes. ......................... 36
4.4 The main loop of the TLVListenerTCP class ................... 38
4.5 BaseTransportMessage class and derived classes. ............. 42
4.6 AbstractRoutingHandler class and derived classes. .......... 42
4.7 Interception of packets in the IP stack using LIBIPQ. ....... 43
4.8 AbstractRoutingHandler interface. ........................... 45
4.9 The main loop of IPQThread. ................................. 45
4.10 AODV Route Request and Route Reply message format. ....... 47
   (a) AODV Route Request. .................................... 47
   (b) AODV Route Reply. ...................................... 47
4.11 AODV extensions. ........................................... 48
4.12 Splitting service discovery payload into different AODV extensions. 48
4.13 The sendMessage() function. ................................. 49
List of Tables

1.1 The four most important components of SIP. .......................... 8
2.1 Combinations of service discovery strategies and transport layers 21
3.1 List of all message types. .................................................. 26
3.2 List of all data types. ....................................................... 27
4.1 Log levels and their corresponding function call. ..................... 41
A.1 Configuration options for the DSD system. ............................ 65
List of Listings

4.1 Example of a configuration file. . . . . . . . . . . . . . . . . . . . 40
4.2 Setting up an iptables rule to intercept AODV routing packets. . 42
4.3 Contents of a registration object. . . . . . . . . . . . . . . . . . 56
4.4 Service identifier for SIP service discovery. . . . . . . . . . . . . 56
4.5 Service payload for SIP service discovery. . . . . . . . . . . . . . 56
5.1 Commands to start the transport handler test. . . . . . . . . . . 59
A.1 Building the project. . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.2 Cleaning up. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.3 Commands used to setup LIBIPQ. . . . . . . . . . . . . . . . . . 66
A.4 Example for starting the DSD system . . . . . . . . . . . . . . . . 66
A.5 Starting the SIP proxy server. . . . . . . . . . . . . . . . . . . . . 67
A.6 Removing iptables rules. . . . . . . . . . . . . . . . . . . . . . . . 67
B.1 EBNF description of the configuration file format. . . . . . . . . 69
B.2 Example of SIP proxy server configuration file. . . . . . . . . . 70
Bibliography


