ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science
Institute for Pervasive Computing
Information and Communication Systems Research Group

Semester Thesis SS06

A Netfilter-Match for AODV-Extensions to be used in Service Discovery/Ad Hoc Networks

Manfred Stock

August 3, 2006

Supervising Assistant: Patrick Stuedi
Supervising Professor: Gustavo Alonso

Abstract

This semester thesis shows the implementation of a Netfilter kernel module which matches packets containing an AODV (service discovery [7]) extension. These packets may then be passed to a kernel queue for access from userspace. The module was tested with a service discovery system and some performance measurements were done.

Contents

1 Introduction

It lies in the nature of Mobile Ad Hoc Networks (MANETs) that there is no centralized infrastructure available - nevertheless, many applications in MANETs depend on the availability of services such as SIP, DNS or SLP. One possible approach to solve this problem is the distribution of the service discovery among the available nodes in the network. In [1] and [7] an implementation of such a distributed service discovery (DSD) framework was proposed. Their work uses piggybacking of service discovery information to routing layer messages and implements a fully distributed SIP service - [4] did the same for SLP. Unfortunately, measurements showed that the normal routing got slower and that the service lookup times were not satisfying, either. It was then suspected that the loss of performance came from the overhead of passing around the routing packets between kernel- and userspace as it happens in DSD. This is the point where this semester thesis comes into play: A netfilter match, matching only those packets which need to pass through the DSD system, would improve the performance of the normal routing because packets would only get passed to userspace in two cases: (1) When they are already involved in a currently running service lookup (and thus containing a service discovery extension) or (2) when they are needed for the piggybacking of service discovery information. This presumably reduces the interference with the normal routing significantly.

This thesis will first give an introduction to Linux netfilter and especially netfilter matches followed by the description of the actual implementation of such a match for the AODV routing protocol. It will then present the results of some measurements with the system from [4] and an improved version which was developed in parallel to this thesis. Finally, a users' guide for the developed match and a conclusion will be presented.

1.1 Distributed Service Discovery (DSD)

This thesis uses a distributed SLP service for testing and for an exemplary integration of the match into a DSD system. The service provides the normal SLP interface over TCP and UDP for the handling of service registrations and lookups but uses a different backend based on the DSD system where service information is efficiently piggybacked to routing messages. The problematic dependence on the underlying routing protocol has been resolved by using routing handler plugins which can be treated like a black box receiving and modifying raw routing packets. Because the system used in this thesis makes use of an AODV plugin and AODV being an on-demand routing protocol, the service discovery also uses a lazy approach to perform service registration and lookup. Hence, the registration results only in the local storage of the service information while a lookup gets piggybacked to an outgoing routing message, in this case using AODV extensions. Exploiting a routing protocols' extensions for service discovery makes perfect sense since routing protocols for MANETs are already optimized for this type of network. Because they are running anyway, some packets can be saved as well since a service discovery request is normally followed by a route request for the host providing the service - and this route already has been established by the route request/reply carrying the service discovery request/reply.

2 Related Work

2.1 Linux Netfilter

The DSD systems introduced in 1.1 make use of netfilters' `libipq` which may be used together with the `iptables` QUEUE target to transfer packets between kernel and userspace. A more thorough introduction to netfilter will be given in 3.

2.2 AODV Routing Protocol

AODV, the *Ad-hoc On-demand Distance Vector* routing algorithm [3] is a reactive routing protocol (it only establishes a new route to a destination on demand) for use in eg. MANETs. For a short description of its packet formats, see 4.1.

This thesis and the aforementioned DSD systems make use of the AODV-UU [8] implementation of the AODV protocol. It consists of a Linux kernel module which uses netfilter to capture data packets and a userspace daemon which maintains the kernel routing table.

2.3 Distributed Service Location Protocol for MANETs

The service location protocol (SLP, [2]) provides a framework for the discovery and selection of network services. An implementation of SLP on top of the DSD system is provided by [4] and was used for testing of the netfilter match and also when doing measurements of the performance.

3 An Introduction to Linux Netfilter

Linux netfilter[1] is the powerful packet filtering framework inside the Linux kernel. It consists of three parts:

- So called "hooks" for each protocol which are points in a packet's traversal of that protocol stack where netfilter gets called.
- The ability for parts of the kernel to register for listening to these hooks and then process packets which traverse the hook. After processing, they tell the kernel what to do with the packet: Discard it, accept it, forget about it or queue it for userspace.
- A queue where packets can be collected for sending to userspace.

The netfilter homepage has some documentation (especially [5] and [6]) which is for Linux 2.4.x but not entirely outdated yet. This is because the packet filtering infrastructure was not completely changed between the 2.4.x kernel series and the current 2.6.x kernels - looking at earlier major releases, it seems that this had been a tradition before: 2.0 used `ipfwadm`, 2.2 `ipchains` and 2.4 `iptables` which is still the case for 2.6. It nevertheless turned out that the most reliable and current documentation is contained in the kernel and `iptables` sources themselves, not only because they contain many working examples.

3.1 Path of a Packet through Netfilter

As the netfilter framework is merely a series of hooks in various points in a protocol stack, a packet selection system called IP Tables has been built over it. IP Tables currently provides a set of up to three tables ('filter', 'nat', 'mangle', depending on kernel modules loaded) containing so called firewall chains which are lists containing rules. The kernel modules providing the tables can register them at the relevant hooks and ask for a packet to traverse a given table. Because this thesis only used the 'filter' table, 'nat' and 'mangle' won't be considered in the following paragraphs (they work similarly, but are registered at different sets of hooks).

Figure 3.1 shows the traversal diagram for the 'filter' table. It contains the built-in chains INPUT (for packets destined to local sockets), FORWARD (for packets being routed through the system), and OUTPUT (for locally-generated packets).

Entries of a chain define criteria to match a packet and a target specifying the next rule which should handle the packet in case the rule matches. This may be a user-defined rule or one of the special values ACCEPT (let the packet pass), DROP (throw packet away), QUEUE (pass the packet to userspace), or RETURN (go to next rule in calling chain). If a packet completely traversed a built-in chain and was neither dropped nor accepted, the default policy of the chain decides it's fate.

The traversal itself looks as follows (see also Figure 3.1):

- When a packets comes in, the kernel first examines its destination and then decides where it will be handled next (routing):
    1. If it's destined for the local system, it passes down to the INPUT chain. In case the packet survives, any processes waiting for it will receive the packet afterwards.

[1] `http://www.netfilter.org/`

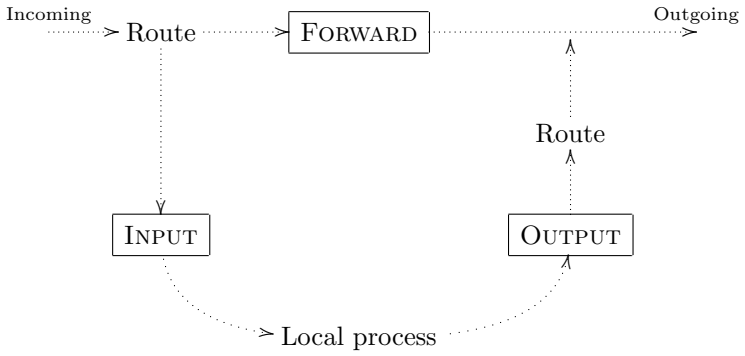

Figure 3.1: IPv4 IP Tables traversal diagram for the 'filter' table

2. If it isn't destined for the local system and forwarding is not enabled or the kernel does not know how to forward the packet, it gets dropped. If forwarding is enabled, and the packet is destined for another network interface, the packet goes to the FORWARD chain - if it survives this, it gets sent out again through the other interface.

- Packets may also be generated locally. Before they leave the system, they go through the OUTPUT chain. If the chain accepts a packet, routing decides on which interface it should be sent and the packet continues out to the interface it is destined for.

3.2 Linux Kernel Modules

Kernel modules are pieces of code which may be loaded into and unloaded from the kernel at runtime. This allows an extension of functionality without requiring a reboot of the system. Netfilter may also be extended this way.

For a module to be loadable into the kernel, it must provide two functions: `int init_module (void)` and `void cleanup_module(void)`. The first function initializes the module when it gets loaded and may eg. register handlers while the latter should undo whatever happened during initialization, as it gets executed when the module is removed from the kernel. But as of kernel version 2.2, these two functions should not be used directly anymore - `module_init(int (*initfn )())` and `module_exit(void (*exitfn)())` are macros from `linux/init.h` replacing them, which permits two things: The functions may have arbitrary names and it allows the module to be either built directly into the kernel or as a loadable module. If it's built into the kernel, the initialization function is being executed at system boot and the cleanup function never at all. There are also two other macros which help save some space: `__init` and `__exit`. They have no effect for loadable modules, but if the modules get built into the kernel, `__init` takes care of discarding the initialization function when it has finished, while the latter omits the cleanup function because a built in module cannot be unloaded.

The initialization function must return zero if everything went well and nonzero otherwise.

3.3 A New Match

The following sections will give a short overview of the structure of a new netfilter match and are based on [5]. Starting with kernel 2.6.16, IP Tables is getting an overhaul and conversion to `x_tables` which should ease the maintenance in the long run. Current kernels therefore contain slightly modified versions of the structs and method signatures mentioned below, but the basic structure is still the same (the main difference being that `ipt_*`-functions and structs now start with `xt_`, but this is taken care of by some preprocessor macros in the kernel source - old sources still compile, albeit with warnings because some method signatures changed; see Listing 3.1 for an

```
static struct ipt_match aodvext_match = {
320   .name           = "aodvext",
      .match          = &match,
322   .checkentry     = &ipt_aodvext_checkentry,
      .me             = THIS_MODULE,
324 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,17)
      .matchsize      = sizeof(struct ipt_aodvext_info)
326 #endif
};
```

Listing 3.1: Definition of `struct ipt_match` from `ipt_aodvext.c`

example of manually working around one difference in the format of a struct between IP Tables in kernel 2.6.8 and 2.6.17+ using a preprocessor directive (older kernels did not have a `matchsize` field in the struct, but newer kernels require this to be set if it is nonzero)).

3.3.1 Kernel Module

New match functions are usually implemented as a standalone module. At the core of a new match function lies a `struct ipt_match` containing primarily the following fields:

list Used for inserting the match into a list, may be set to any junk, normally `{NULL,NULL}`.

name Name of the match function, should match the name of the module as this is used for autoloading of the module from userspace.

match Pointer to a match function which gets the packet and returns a nonzero value if the packet matches.

checkentry Another pointer to a function which checks the specifications for a rule. Returns nonzero if the rule is accepted from the user. This way one can disallow the use of a match built for UDP in a rule that allows TCP packets.

destroy Points to a function as well and gets called when an entry using this match is deleted. Used to clean up resources allocated in `checkentry`.

me Set to `THIS_MODULE` thereby pointing to the module. Mainly used for reference counting.

matchsize Size of the data structure used for the configuration of the match. Required if nonzero for kernels $\geq$ 2.6.17 - see Listing 3.1 for an example.

This struct will be passed to `int ipt_register_match(struct ipt_match *match)` during initialization of the module and `void ipt_unregister_match(struct ipt_match *match)` during cleanup when unloading the module. Listing 3.1 shows an example for this struct from the implementation of the match. As one can see, not every field must be given a value, only those which are really used.

Userspace $\Leftrightarrow$ Kernel Communication

One may register new socket options for communication between a match (or netfilter modules in general) and userspace. This is done by setting up a `struct nf_sockopt_ops` in a way that is similar to the one for `struct ipt_match` from 3.3.1 by defining ranges for the options and corresponding `get` and `set` function pointers. Like before, this struct then gets registered using `int nf_register_sockopt(struct nf_sockopt_ops *opts)` and unregistered by `void nf_unregister_sockopt (struct nf_sockopt_ops *opts)`. After loading the module, the defined socket options may be modified or read from userspace by calls to `setsockopt()` or `getsockopt()`, respectively, on a raw socket.

3.3.2 Shared Library

A shared library must be provided to use a new match with the `iptables` command. The name of the match is being passed to `iptables` by the `-m name` command line option. `iptables` then searches for a library called `libipt_name.so` in its library directory and opens it. The shared library must execute `void register_match(struct iptables_match *me)` during initialization to register the match. A `struct iptables_match` mainly contains the following fields:

name The name of the match function - it should match the library and module name.

help Function printing the options synopsis.

init May be used to initialize extra space of the `ipt_entry_match` structure which will be available in the kernel as well and is used to pass the options to the module.

parse This function gets called when an unknown command line option is seen and should return nonzero if the option was for this library.

print Creates the match-specific output which appears when `iptables -L` gets called.

save Reverse of parse: Print the options which were used to create the rule.

extra_opts A `struct option[]` containing the definition of options for the match. This will be merged with the current options and passed to `getopt_long()`.

size Like matchsize inside the kernel module, but should be set using the `IPT_ALIGN()` macro to ensure correct alignment.

The association between the kernel module and the match gets established through the name fields of the `struct iptables_match` of the library and the `struct ipt_match` of the module. If the module shall automatically be loaded when the match is used in a rule, the same name must also be put into the name of the module (that is, if the match is called “aodvext”, then the module must be in a file named “ipt_aodvext.ko”).

4 Implementation of the Match

In this chapter, an implementation of the netfilter match for AODV extensions will be presented. This starts with an introduction to the AODV message formats because they will need to be parsed to some extent when looking for extensions.

The netfilter match consists basically of the following three parts:

- A kernel module which, among other functions, matches AODV messages containing an extension.
- A shared library needed by the `iptables` command for accessing the functionality of the kernel module.
- An interface to interact with the module from userspace when the module is loaded and participating in an IP Tables rule.

4.1 AODV Message Formats

AODV [3] has four different message formats, but only two of them were relevant in this thesis: Route requests and replies. The other two are route error messages and route reply acknowledgements. AODV also supports attaching extensions to routing messages, which is being used in eg. [1] and [7]. The following sections will give a short description of these message formats.

4.1.1 Route Request (RREQ)

See Table 4.1 for the message format. Route requests have type 1. The hop count ist the number of hops from the originator IP address (the node which originated the route request) to the node handling the request, and the destination IP address is the IP address of the destination for which a route is desired.

0								1								2								3		
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Type								J	R	G	D	U	Reserved													Hop Count					
RREQ ID																															
Destination IP Address																															
Destination Sequence Number																															
Originator IP Address																															
Originator Sequence Number																															

Table 4.1: Route request (RREQ) message format

4.1.2 Route Reply (RREP)

See Table 4.2 for the message format. Route requests have type 2. The meaning of the other fields is basically the same as for a route request. Additionally, the lifetime signifies the time (in milliseconds) for which nodes receiving the route reply consider the route to be valid.

0 1 2 3 4 5 6 7	8	9	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 0 1
Type	R	A	Reserved	Prefix Sz	Hop Count
Destination IP Address					
Destination Sequence Number					
Originator IP Address					
Lifetime					

Table 4.2: Route reply (RREP) message format

Hello Messages

Hello messages may periodically be broadcasted to offer connectivity information to neighbouring nodes. These messages are route replies with a TTL of 1, a hop count of 0 and the destination address set to the IP address of the node sending the message. The destination sequence number is additionally set to the node's latest sequence number and the lifetime has a defined value as well, but these two fields are not relevant here.

4.1.3 Extensions

RREQ and RREP messages may have extensions which are attached after the message data. The format of such extensions is shown in Table 4.3. The type field contains values between 1 and 255, and the length field constitutes the length of the type-specific data, not including the type and length fields of the extension, in bytes.

0 1 2 3 4 5 6 7	8 9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Type	Length	type specific data...

Table 4.3: AODV extensions format

4.2 Matching AODV Extensions

The routing messages of AODV are encapsulated in UDP packets which are sent to port 654, so matching of simple messages is straightforward assuming no other packets are sent to this port. It becomes a litte more involved when only those packets containing extensions should be matched. Because extensions may be attached to RREQ and RREP messages which have different lengths it's necessary to first find the type of the message. As soon as the type is known, one can simply check if the length of the packet is more than what would be expected for a packet of the corresponding type. If this is the case, it's fairly likely that the packet indeed contains extensions (if it doesn't, it's no valid AODV packet anyway).

4.3 Kernel Module

The module can work in one of two modes: It either matches packets containing AODV extensions or the match can be turned on and off from userspace without changing the rule. These modes are mutually exclusive: Only one mode per rule can be used. If the state of the switchable mode gets changed, this does not have an influence on any rule using the match in the extensions mode, but it will modify the behaviour of every rule using the match in the switchable mode.

All functionality is implemented in the file `ipt_aodvext.c` which includes the header file `ipt_aodvext.h` (see Listing 4.1). The header file defines a `struct ipt_aodvext_info` which is used to pass configuration information from the shared library to the match: It should be initialized in the

```
#ifndef _IPT_AODVEXT_H
#define _IPT_AODVEXT_H

/* Switch on/off */
#define RS_ON +1
#define RS_OFF -1

/* Number of socket option */
#define RS_SOCKOPTNUM 3141592

struct ipt_aodvext_info {
  u_int8_t invert;
  u_int8_t aodvext;
  u_int8_t switcher;
  u_int8_t alldebug;
  u_int8_t matchdebug;
  int16_t hello;
};

#endif
```

Listing 4.1: `ipt_aodvext.h` header file

library when adding a new rule and may then be accessed from within the `int match()` function in the kernel module.

4.3.1 AODV Extensions Mode

In this mode, only packets which presumably contain AODV extensions are matched. The way this works is as follows:

- Check if the packet could be a valid AODV packet: This is done by partially parsing the UDP packet and checking that the packet is large enough to even contain an AODV message.
- Get the AODV message type and check if the size is large enough such that it is possible that AODV extensions have been appended to the message. If this is the case, the packet matches.

4.3.2 Switchable Mode

This mode may be influenced from userspace through a new socket option (see 3.3.1) which gets registered by this module. The definition of the socket option number may be found in Listing 4.1. This way, one can turn matching on and off which may be used by the DSD system to match packets only if they are needed for piggybacking service discovery information. It works by changing the internal state of the module: When enabled, the `int match()` function returns a nonzero value (which means that the packet matched), when disabled, the function returns 0. The advantage of this is that the rule does not have to be removed from/inserted into the chain every time, so this is presumably more efficient.

In addition to the switchability, this mode may also be configured to match every n'th hello message going through it even when the match was turned off - this could be used by the DSD system to freely transport service discovery information to it's neighbours.

4.3.3 Logging

The module also contains some code to provide logging output which can be enabled when adding a rule to an IP Tables chain. It either prints information for every AODV message which is

```
E:AODV RREQ: SRC: 192.168.0.9; DST: 192.168.0.4; Hopcount: 0; DST-IP:
    127.0.0.1; DST-SEQ: 1153906252; ORIG-IP: 192.168.0.9; ORIG-SEQ:
    1153906252; RREQ-ID: 1153906252; Has extension: 1
T:AODV RREQ: SRC: 192.168.0.4; DST: 172.16.0.3; Hopcount: 0; DST-IP:
    127.0.0.1; DST-SEQ: 1153906339; ORIG-IP: 192.168.0.4; ORIG-SEQ:
    1153906339; RREQ-ID: 1153906339; Has extension: 1
T:AODV RREQ: SRC: 192.168.0.4; DST: 172.16.0.3; Hopcount: 0; DST-IP:
    127.0.0.1; DST-SEQ: 1153906931; ORIG-IP: 192.168.0.4; ORIG-SEQ:
    1153906931; RREQ-ID: 1153906931; Has extension: 0
E:AODV RREP: SRC: 192.168.0.9; DST: 192.168.0.4; Hopcount: 0; DST-IP:
    127.0.0.1; DST-SEQ: 1153907351; ORIG-IP: 192.168.0.9; Lifetime: 65536;
    Has extension: 1
```

Listing 4.2: A few examples of the logging output

```
void __attribute__ ((constructor)) my_init(void)
{
  register_match(&aodvext);
}
```

Listing 4.3: Library initialization from `libipt_aodvext.c`

processed or only for those that matched. This could be useful when debugging problems on higher levels of the system.

The output mainly consists of details from the AODV message. An example of such an output may be found in Listing 4.2. The first letter denotes why the packet matched: `E` stands for extension, `T` for the switchable mode and `H` for hello message. An `A` will be printed if all AODV messages passing through the match shall be logged.

4.4 Shared Library for `iptables`

As mentioned in 3.3.2, a shared library for the use with `iptables` must be provided. The netfilter hacking HOWTO [5] mentioned that the library should contain an `_init()` function which gets automatically called when loading the library. This did not work together with `libtool`, so some research lead to the solution in Listing 4.3. According to [9], this solves another problem as well: `_init()` together with `_fini()` have been obsoleted and their use could even lead to unpredictable results.

4.5 Interaction With Module From Userspace

According to 4.3.2, the module registers a new socket option to disable/enable the match when the match is in switchable mode. The simple code snippet in Listing 4.4 is an example of using this socket option to first enable and then disable the match again. It finally reads the current value of the option.

This code can also be encapsulated in a C++ class to make it accessible by object oriented means from the DSD system which is implemented in C++. See 4.6 for more details on this integration.

An interesting point about the implementation of this feature is that it was implemented using a counter: If the match is enabled n times, it must be disabled n times until it is turned off again. The reason for this is to allow concurrent access from different processes - if one process disables the match only as many times as it enabled it before, it cannot disable the match for another process which might still need it to be active. Because this concurrent access could lead

```
int sock,val;
int size = sizeof(int);

if((sock = socket(AF_INET,SOCK_RAW,IPPROTO_RAW)) == -1){
  perror("socket");
  exit(1);
}

val = RS_ON;
if(setsockopt(sock, IPPROTO_IP, OPTNUM, &val,sizeof(int())) == -1){
  perror("setsockopt");
}

val = RS_OFF;
if(setsockopt(sock, IPPROTO_IP, OPTNUM, &val,sizeof(int())) == -1){
  perror("setsockopt");
}

if(getsockopt(sock, IPPROTO_IP, OPTNUM, &val,&size) == -1){
  perror("getsockopt");
}
```

Listing 4.4: Using the socket option

to problems with the counter in the kernel, all write accesses to the counter must be protected by locking. This can be implemented by defining a mutex using `static DECLARE_MUTEX(accept_lock)` in the module and then calling `down_interruptible(&accept_lock)` to acquire the lock and `up(&accept_lock)` to release it again after modifying the counter. This way, a simple reference counter was implemented, which turned out to be useful later on.

4.6 DSD-SLP Integration

As this match was written for use in the DSD system, it finally had to be integrated into the SLP daemon which is built on top of the DSD system. See Figure 4.1 for an overview on how the netfilter match and the SLP daemon work together.

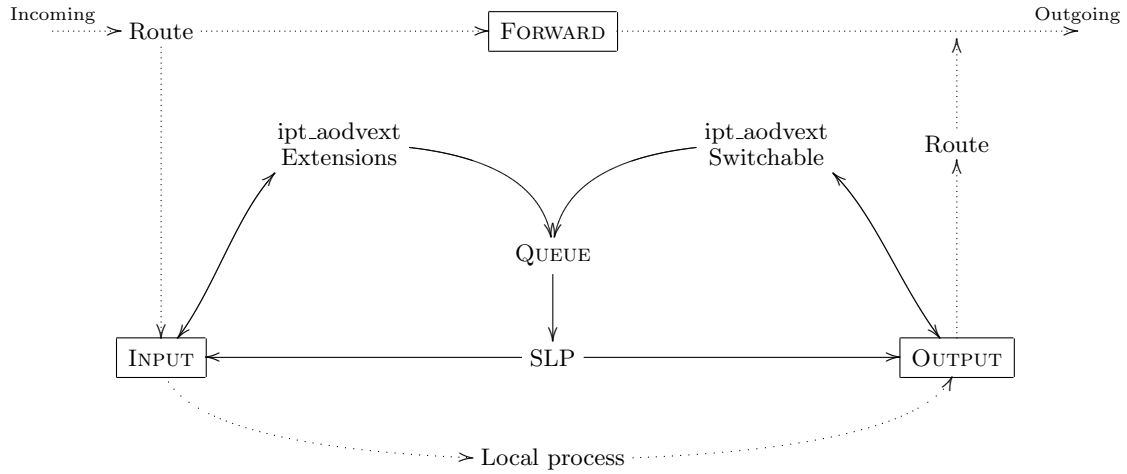

Figure 4.1: Overview on integration of SLP daemon and netfilter match

```
iptables -A INPUT -p udp --dport 654 -m aodvext --aodvext -j QUEUE
iptables -A OUTPUT -p udp --dport 654 -m aodvext --switcher -j QUEUE
```

Listing 4.5: Minimal commands to set up the rules

```
class AODVRuleSwitcher
{
private:
        int sock;
        void set(int val);
public:
        AODVRuleSwitcher();
        ~AODVRuleSwitcher();

        void enable();
        void disable();

        int value();
};
```

Listing 4.6: `AODVRuleSwitcher.h` with the interface for `AODVRuleSwitcher`

It basically functions as follows: There are two IP Tables rules - one in the INPUT and one in the OUTPUT chain. Both rules make use of the match and have QUEUE as their target - see Listing 4.5 for a minimal variant of these rules. The rule in the INPUT chain matches packets containing an AODV extension and the one in the OUTPUT chain matches only AODV messages when activated or if the message is a hello message and these shall also be matched (which is the default). All the matching packets are queued up for userspace processing and may be read using `libipq`. This is the point where the SLP daemon jumps in: It reads the packets from the queue and processes them. For a detailed description of this processing, see [1].

Simplified, it works as follows: After receiving a service request, the system first checks if the service information is available locally. If this is the case, the request gets answered immediately. If not, the system triggers routing messages by sending a packet to a nonexistent address of the subnet which results in a route request from the AODV routing daemon. This RREQ message needs to be intercepted to attach the service discovery request: Hence the rule in the OUTPUT chain must be activated. After attaching the service discovery extension, the packet is sent to the network and will be handled like any other route request. However, if a receiving node is running the SLP daemon and the rule in the INPUT chain is set, the packet gets intercepted and put on the queue for processing by the SLP daemon because it contained an extension. If no local information about the requested service is available, the message will just be ignored. Otherwise, the message is being modified to look as if it was a route request for the local node, which will trigger a route reply from the AODV routing daemon. Again, this RREP needs to be intercepted for attaching the information about the service, thus the rule in the OUTPUT chain must be activated.

As one can see, the SLP daemon needs to keep track of service requests and their corresponding, explicitly or implicitly triggered, routing messages. It does so by maintaining `AODVContext`s in an `AODVContextStore`. Because it also needs the capability to control the rule in the OUTPUT chain, an `AODVRuleSwitcher` class was implemented for the userspace $\Leftrightarrow$ match interaction (see Listing 4.6 for the interface). It basically works as described in 4.5 and therefore makes use of the reference counting implemented by the match.

Due to the strong correlation between the rule in the OUTPUT chain and the context store (as long as the store is not empty, outgoing routing messages need to be intercepted), it would have been quite natural to do the whole integration in the `AODVContextStore` class: When storing a new context, the match could be "activated" and when removing a context, it could be "turned

off" again. Unfortunately, the `RoutingHandlerImplAODV::triggerMessageAndAddContext()` method triggered a message before adding a context to the context store. Because this could result in "lost" routing messages (the rule may not be activated fast enough), an `AODVContextStore::enableRule()` method was introduced which is used in the `RoutingHandlerImplAODV` to activate the rule before triggering messages.

5 Results of Measurements

Two series of measurements were done: One with the current DSD-SLP implementation which was mostly developed in parallel to this thesis and because the results were fairly surprising, a subset of the measurements was also repeated with the old implementation from [4] and basically resulted in what was expected.

The measurements were done using 6 notebook computers running Debian 3.1 (Sarge). Five of them had a 2.0Ghz Mobile Pentium 4 and were equipped with an integrated 11Mbit/s IEEE802.11b wireless network interface card and were running kernel 2.6.8 while the sixth had a Pentium M processor with 1.73GHz combined with a 54Mbit/s IEEE802.11g wireless network interface card (used at 11Mbit/s through configuration) and was running kernel 2.6.11. They were controlled using `ssh` over ethernet from a desktop workstation and the measurements were done over a wireless LAN in ad hoc mode. Because it would be difficult to find a spacial separation of the notebooks which would have required multihop communication between them, an artificial separation using IP Tables rules was done: They were only allowed to communicate with their direct neighbours (and the workstation over ssh), all other traffic was dropped by the default INPUT policy.

All measurement values appearing in this section are the results over a set of 10 tests. The routing performance was measured by restarting the AODV daemon and issuing an ICMP packet to the target host afterwards, so the time required for the route discovery could be calculated from entries in the log of the AODV daemon. The same goes for the service lookup times where the SLP daemons were restarted after every lookup.

When looking at the graphs (see Figure 5.1 and Figure 5.2), two special things can be observed: (1) Route discovery over 1 hop always takes 0ms and (2) there is a jump in the time taken for route discovery between 3 and 4 hops. The reason for this lies in the underlying AODV routing protocol: (1) Direct neighbours always know the route to each other because of the regularly sent hello messages and (2) because the AODV protocol should use an expanding ring search (see [3, 6.4]), the TTL gets increased over time. Since AODV-UU starts with a TTL of 2, the RREQ over 4 hops only reaches the node with distance 2 from the destination which does not know a route to the destination. Therefore, AODV waits `RING_TRAVERSAL_TIME` milliseconds for an RREP (again, according to [3, 6.4]) and then sends another RREQ with an increased TTL[1]. The SLP daemon does not implement the optimization of sending service information to it's neighbours yet, so there the jump in the lookup time happens already one hop earlier.

5.1 Old DSD-SLP Implementation

See Figure 5.1 for the results of the series of measurements with the DSD-SLP implementation from [4]. In contrast to the measurements in [7], debugging output of the SLP daemon was not turned off, so the results in this thesis were expected to be slightly worse than theirs, which actually was the case. As one can see, the running SLP daemon which intercepts all AODV messages imposes a significant delay on the routing itself: The time for a route discovery is often more than tripled.

This was clearly improved by using the match implemented in this thesis. Because the system used for these tests was already deprecated, the match was only used in the INPUT chain since it would not have made much sense to integrate the control of the match in the OUTPUT chain into an obsolete system. But one can see that the delay is about halved, because only outgoing

[1] Looking at the AODV-UU [8] source shows that `RING_TRAVERSAL_TIME` $= 2 \cdot$ `NODE_TRAVERSAL_TIME` $\cdot$ (`TTL_VALUE` + `TIMEOUT_BUFFER`) with `NODE_TRAVERSAL_TIME` $= 40$ and `TIMEOUT_BUFFER` $= 2$ - so for a TTL of 2, this results in 320ms which matches the measured results. If no RREP was received, the TTL gets always increased by 2.

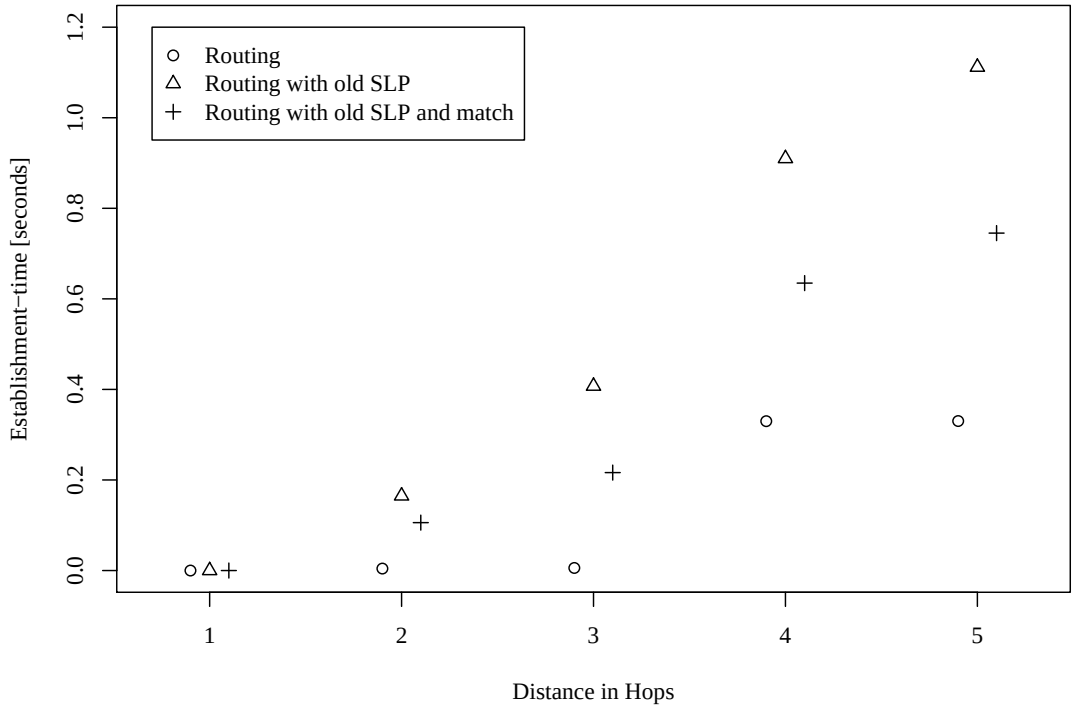

Figure 5.1: Results from experiments with the old SLP implementation

packets get delayed by passing through the SLP daemon - it might therefore be assumed that the integration of the match would result in similar improvements, which would nearly reduce the delay to the routing only case.

5.2 Current DSD-SLP Implementation

See Figure 5.2 for the results of the series of measurements with the current DSD-SLP implementation. One can clearly see why the results were called surprising in 5: There is hardly any noticeable difference between normal routing, routing with active SLP daemon (where all packets pass through) and active SLP daemon together with the netfilter match (where packets only pass through the daemon when necessary). Because it was obvious that these improvements had nothing to do with the match, the measurements in 5.1 were done to get a clue of the reason since there were several possibilities:

- This thesis used 2.6.x kernels while [7] used the 2.4.x series. As there were many (performance related) improvements between these two major releases, this could have made a difference.
- Some minor difference in configuration (IP Tables rules, network settings, environment, AODV implementation) could have occurred.
- The improved DSD-SLP implementation might have fixed a rather fundamental problem present in the old implementation.

It seems that the most likely reason is the improved DSD-SLP implementation because the results with the old implementation fit the ones from [7] pretty well. This gets also supported by the

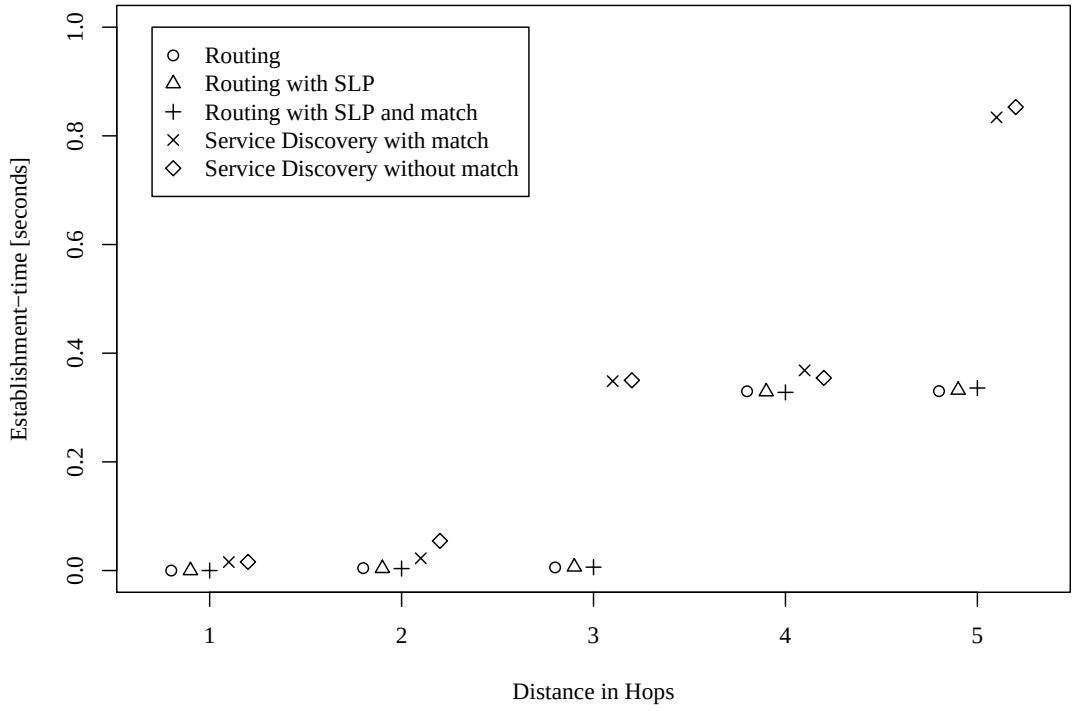

Figure 5.2: Results from experiments with the current SLP implementation

improved service lookup times which are only slightly worse than the routing itself.

6 Users' guide

The following sections will give a few hints on compiling and using the IP Tables match developed in this thesis. There will also be some pointers to helpful applications which were used and some scripts which were developed during the work on this thesis.

6.1 Compilation and Installation

6.1.1 Platform Requirements

The following software setup has been tested to work with the match for compilation and usage:

- GNU/Linux operating system with kernel 2.6.x (really tested with .8, .11, .16 and .17, but others should work as well whereas newer ones might require minor modifications for the compilation of the module).
- GCC 3.3/4.0 (other versions should work as well - if the kernel could be/was compiled with a given version, the match ought to compile, too).
- `iptables` $\geq$ 1.2.11 including `libipq`.
- `libtool` $\geq$ 1.5.6
- `autoconf` $\geq$ 2.13
- `automake` $\geq$ 1.9.5
- AODV-UU $\geq$ 1.9.1

There might be some additional dependencies coming from the DSD system in which the match should be used. A few auxiliary requirements might arise when all the provided scripts shall be used, such as Perlipq (`IPTables::IPv4::IPQueue`) in case of the `ipq_aodv_display` script.

Developement and testing was mostly done on Debian GNU/Linux 3.1 (Sarge) little endian systems, but some minor tests showed that the match should also work on big endian machines and more recent GNU/Linux distributions (namely Debian Testing (Etch)). But basically any GNU/Linux distribution fulfilling the aforementioned requirements should be usable.

6.1.2 Build Process

The build process is based on the well known so called GNU Autotools (`autoconf`, `automake`, `libtool`), so the build is quite straightforward. If the `configure` script et al. do not exist yet, the `autogen.sh` script must be executed first. Else or afterwards, respectively, the famous triple `./configure && make && make install` should configure, compile and install the module and the shared library together with the `ruletoggle` and `aodv_gen` tools. Because the module and the match need to be put into system directories, at least the last command must be executed as `root`. If the installation shall ever be undone, an `uninstall` target was created in the `Makefile`.

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
QUEUE      udp  --  anywhere             anywhere            udp dpt:654
    AODV extensions

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
QUEUE      udp  --  anywhere             anywhere            udp dpt:654
    ruleswitcher: 0 hello: 0'th
```

Listing 6.1: Output of `iptables -L` for rules from Listing 4.5

6.2 Usage

The user interface to the match is the normal `iptables` program which got extended by the shared library. Calling `iptables` with the `-m aodvext` option will make it load the library and the match-module. `iptables` then also accepts other options for configuring the match which will be described in the next section.

6.2.1 `iptables`-Options

The following options are available:

`--invert` Invert the result, for example match only packets without extension.

`--aodvext` Match only packets with extensions - the default mode. See 4.3.1 for details on this mode.

`--switcher` Allow userspace to turn on/off the match using the socket option. See 4.3.2 for details on this mode.

`--hello <n>` Match every n'th Hello-message if in switcher-mode. The default is 0 which matches all Hello messages, -1 would turn off matching of Hello messages.

`--alldebug` Print information on every processed packet into the kernel log. See 4.3.3 for details.

`--matchdebug` Similar to `--alldebug`, but this only prints information on packets that actually matched.

The options `--aodvext` and `--switcher` can not be combined because they decide in which mode the match shall run. If this is done nevertheless, the last appearance of one of these options will be used. Calling `iptables -m aodvext -h` will give a short summary of the available options for the `aodvext` match. See Listing 4.5 for a short example of the usage. The use of these two rules would result in the `iptables -L` output of Listing 6.1 which shows the mode the match operates in for the rule, the reference counter of the switcher mode and if Hello messages shall be matched. If one of the debug options had been specified, this would be indicated by an `ad` or `md` flag, respectively.

Because the match only works with UDP packets, it always requires the `-p udp` option. If the call to `iptables` fails (for example with `iptables: Invalid argument`), a look at the output of `dmesg` might be helpful.

6.3 Useful Tools

Because testing of the match and the measurements involved several computers (up to 6 notebooks), one may need some tools to work with them efficiently. In the following sections, a couple of useful scripts and tools for this task will be presented.

6.3.1 ClusterSSH

ClusterSSH[1] allows one to control multiple ssh sessions at the same time. It opens a `xterm` for each session and distributes them on the screen, allowing the interaction with individual terminals if desired. The hosts may be given on the command line or added at runtime. A configuration file is also supported which allows the definition of so called clusters - this is useful if one wants to connect to different sets of hosts.

6.3.2 Synergy

Synergy[2] allows sharing of a single keyboard and mouse between multiple computers with different operating systems, each with its own display. This way one can "simulate" a multihead setup when using two or more computers or notebooks. Synergy uses a configuration file to define the setup and may also be used over a port forwarded by `ssh`.

6.3.3 Scripts and Tools

During the developpement of the match, a set of scripts and tools have been written to ease the work. They are not very generic, but some of them use a simple configuration file where a layout for the desired network can be defined. The following sections will provide some information on the most useful scripts and their usage. Most of the scripts will use `ssh` to connect to hosts defined in the configuration file, so it might be useful to set up public key authentication to save typing the password for every command executed (because the scripts use different users, this is necessary for `root` and the user used for testing (whose name currently is often hardcoded into the scripts...)). Many scripts also expect the configuration file to be in the same directory as themselves with the name `hosts`, but some may accept the path to the configuration file as their first or second argument or even with a `--config` option. Some scripts actually assume that required files may be found under the `~/voicehoc/` directory structure. The best way if something does not work as it should is probably to have a short look at the scripts themselves - they are not very complex and written in Bash or Perl.

Another good idea is the setup of `sudo` - because it is often necessary to execute commands as `root` while working as another user, one can also save some password typing by adding a line like `username ALL=(ALL) ALL` to `/etc/sudoers` using the `visudo` command as `root`. Afterwards, if one needs to execute a command as `root`, one simply prefixes it by `sudo` - it will then ask for the users password (which only happens about once every 15 minutes, depending on configuration) and executes the command as `root`. Some of the scripts presented below actually do make use of `sudo`, so this might be needed for them to work properly.

Configuration File

See Listing 6.2 for an example of the configuration file. The lines at the top define different systems and their MAC addresses while the `admin` line defines the system which will be used for administration and therefore is allowed ssh access. `connection` signifies the network and `prefix` will be used as a prefix for the addresses in the wireless network - the last byte is taken from the number in the hostname (which is an evidence for the claim that these scripts are not very generic).

[1] `http://clusterssh.sourceforge.net/`
[2] `http://synergy2.sourceforge.net/`

```
wllap11=00:02:2D:7B:89:B4
wllap14=00:02:2D:BA:11:93
wllap15=00:02:2D:BA:11:73
wllap6=00:02:2D:7B:88:D8
wllap13=00:02:2D:7B:89:6F
wllap10=00:13:CE:8C:78:D5
admin=172.25.3.244
connection=wllap11-wllap14-wllap15-wllap6-wllap13-wllap10
prefix=192.168.0
```

Listing 6.2: Configuration file for some scripts

aodvadmin

This script will connect to all the hosts defined in the config file to control the AODV daemon. Depending on the name this script gets called with (by using for example symbolic links), it does different things:

`aodvstop` Send a `kill -9` to the `aodvd` instance thereby killing it.

`aodvstart` Start the `aodvd` daemon.

`aodvrestart` Kill and immediately start the AODV daemon again. Useful to clear routes when measuring the route discovery performance.

ipq_aodv_display

Because running the complete DSD-SLP system when testing the match would have been a little inefficient, the `ipq_aodv_display` script has been written. Since it accesses `libipq`, it must be run as `root` and the `IPTables::IPv4::IPQueue` Perl extension must be available. It reads packets from the kernel queue like the DSD-SLP daemon would, and parses them. Some of the information in the packet then gets displayed and the `--printcontent` option would cause the script to also print a hexdump of the content of extensions. An example of the output generated by this script may be found in Listing 6.3. The packet which is being displayed in this output is not a "real" AODV packet, but one generated by the `aodv_gen` program which will be explained later. Because the script returns the packets using the `NF_ACCEPT` verdict to IP Tables, they do not get lost (DSD-SLP does the same).

rulesetup

Since it would be tedious to connect to every host for setting up the IP Tables rules by hand, the `rulesetup` script was implemented: It does this and also accepts some options to control the rules:

`--config=<file>` Path to the configuration file.

`--clear` Clear the IP Tables rules on all hosts and also reset the default policy (which is useful since it got set to DROP on the INPUT chain - which efficiently locks one out of a machine when simply executing `iptables -F`...).

`--noaodv` Do not set rules involving the AODV protocol.

`--oldin` Use the old INPUT chain rules which match all UDP packets on port 654 and put them into the queue.

`--oldout` Same as `--oldin`, but for the OUTPUT chain.

`--hello=<n>` Match every n'th Hello message.

```
Received a new packet with id 3832468768 from NF_IP_LOCAL_IN-hook:
        Datalength:  56
        Trying to parse IP-header:
                Header length: 5 words
                TTL: 64; Protocol: 11 (udp)
                Source address: 127.0.0.1
                Destination address: 127.0.0.1
        Trying to parse UDP-header:
                Souceport: 32780 Destinationport: 654
                Length of UDP-Packet: 36
        Trying to parse AODV-packet:
                Type: 1 (RREQ)
                Hopcount: 0
                RREQ-ID: 1154334716
                Destination address: 127.0.0.1, sequence number:
                    1154334716
                Originator  address: 127.0.0.1, sequence number:
                    1154334716
                Packet seems to contain (an) AODV-extension(s)! Try to
                    parse...
                        Extension 0 is of type 129 and has length 2
                        00 00
                            ..
```

Listing 6.3: Example output of `ipq_aodv_display --printcontent`

`--debug` Use the LOG target to log packets before they pass through a relevant rule.

`--matchdebug` Use the internal logging output to log information on all AODV messages which matched.

`--alldebug` Same as `--matchdebug`, but log all packets passing through the match.

Because only neighbouring nodes should be capable to communicate with each other, the script sets up rules such that only packets coming from the MAC address of a neighbour are accepted, everything else just gets dropped. An example of the rules set on a host in the middle (therefore having two neighbours) can be found in Listing 6.4. One can see that `eth0` is used for administration (port 22 is open for the administrative host) and that the main communication happens over `eth1`, which is the wireless interface.

slprunner

This script first changes to the DSD-SLP daemon directory and then creates another script to execute the daemon. It also takes the default configuration file and adapts it to the subnetwork the script is running in (that is, it changes the pool address range, removes some unnecessary entries and updates timeouts). Finally, it executes `sudo` to run the created script as `root`. Now service registrations and lookups are possible.

wlansetup

The `wlansetup` script sets up the wireless lan using `iwconfig` and `ifconfig`. It sets the ESSID to "aodvext", activates Ad Hoc mode, turns off encryption, configures the channel to be 10 and sets the rate to 11Mbit/s. Afterwards, it sets the IP address to one of the 192.168.0.0/24 subnet by taking the last byte from numbers in the hostname (which therefore should be of the form `wllap11` with 11 being unique for the given network).

Running this script on all available hosts will result in a wireless Ad Hoc network with identical settings on all hosts, but different and unique IP addresses.

```
Chain INPUT (policy DROP 6 packets, 1144 bytes)
 pkts bytes target  prot opt in    out   source         destination
    0     0 ACCEPT  tcp  --  eth0  *     172.25.3.244   0.0.0.0/0        tcp spt:22
  430 53252 ACCEPT  tcp  --  eth0  *     172.25.3.244   0.0.0.0/0        tcp dpt:22
    0     0 ACCEPT  all  --  lo    *     0.0.0.0/0      0.0.0.0/0
    0     0 QUEUE   udp  --  *     *     0.0.0.0/0      0.0.0.0/0        udp dpt:654 MAC 00:02:2
        D:7B:89:B4 AODV extensions
    0     0 QUEUE   udp  --  *     *     0.0.0.0/0      0.0.0.0/0        udp dpt:654 MAC 00:13:
        CE:8C:78:D5 AODV extensions
    0     0 ACCEPT  all  --  eth1  *     0.0.0.0/0      0.0.0.0/0        MAC 00:02:2D:7B:89:B4
    0     0 ACCEPT  all  --  eth1  *     0.0.0.0/0      0.0.0.0/0        MAC 00:13:CE:8C:78:D5
    0     0 ACCEPT  all  --  eth1  *     192.168.0.14   255.255.255.255

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target  prot opt in    out   source         destination

Chain OUTPUT (policy ACCEPT 128 packets, 20204 bytes)
 pkts bytes target  prot opt in    out   source         destination
  342 64827 ACCEPT  tcp  --  *     eth0  0.0.0.0/0      172.25.3.244     tcp spt:22
    0     0 ACCEPT  tcp  --  *     eth0  0.0.0.0/0      172.25.3.244     tcp dpt:22
    0     0 ACCEPT  all  --  *     lo    0.0.0.0/0      0.0.0.0/0
    0     0 QUEUE   udp  --  *     *     0.0.0.0/0      0.0.0.0/0        udp dpt:654
        ruleswitcher: 0 hello: 0'th
```

Listing 6.4: Rules created by a call to `rulesetup` and displayed by `iptables -L -n -v`

ruletoggle

This simple program may be used to toggle the rule when in switching mode (see 4.3.2 for details). If called with a numeric, ≥ 0 argument, it will set the reference counter to the given value. It may also be used as a working example for using the socket option.

aodv_gen

Because the normal AODV routing messages do not contain extensions and the use of the DSD-SLP daemon would have been a little too complex for testing of the match, a simple program which is capable of creating sample AODV messages with or without extensions has been written. It basically sends packets looking like real AODV messages but with fake content to any given host. It takes some options and is used as follows: `aodv_gen [options] <receiver>` where receiver is the IP address or hostname of the host to which the packet should be sent and for `options`, the following values are possible:

`--next <number of extensions to send>` The number of extensions that shall be attached. Extensions are always 2 Bytes long containing 0's as value (which results in 32Bit words which were easy to generate, therefore more "complex" extensions are not possible - but this was enough for testing). Default is 0.

`--rreq` Generate route request messages. The default if no option for the type was given.

`--rrep` Generate route reply messages.

`--type <type to give to extensions>` The type for extensions defaults to 129, with this option this may be changed to an arbitraty value.

`--port <port to send packet to, default 654>` The port where the packet should be sent to. 654 is the default AODV port, but for testing the use of another port might be useful (to for example not mess too much with a running AODV daemon...).

Others

There are a few more scripts which might be useful, but are not that interesting:

`buildAll` Compiles and installs all programs under `~/voicehoc/` which are necessary to test the match as well as the DSD-SLP daemon (it does not compile/install AODV or OpenSLP's `slptool` which are also required - but this must only be done once while the other programs change more often because they may even be work in progress).

`pingtest` Reads the configuration file and tries to send an ICMP message to all hosts in the configuration file. May be used to test if the IP Tables rules cleanly separate the nodes and if the connections get established after starting the AODV daemon.

`sourcesync` Uses `rsync` to synchronize a directory on the administrative machine to all the nodes used for testing. It supports the following options:

- `--config` Where the configuration file with the hosts may be found.
- `--src` Source directory on the local system. Default is `$HOME/voicehoc/benchmarksources/`.
- `--dst` Destination directory on the remote system. Default is `/home/msmeasure/voicehoc/`.
- `--remove` Remove files which do not exist locally - useful when synchronizing fresh sources as it removes the files and directories created during compilation so one may start with fresh sources again.

`syslogfilter` Reads the system log file from stdin and parses it to some extent, colorizing and compressing lines created by the IP Tables LOG target entries from the `rulesetup` script when run with the `--debug` option.

`syslogaodvfilter` Similar to `syslogfilter`, but suited to the `--matchdebug` or `--alldebug` options of the match and `rulesetup`.

7 Conclusion and Future Work

The measurements in 5.1 clearly show that the implementation of a netfilter match can make a difference in performance when not all packets need to pass through userspace anymore. On the other hand, the results from 5.2 with the current SLP implementation imply that the main problem causing bad performance was not at the lower levels where packets got passed around between kernel- and userspace, but rather in the implementation in userspace. This demonstrates once more that sometimes high level improvements might be more efficient than low level optimizations.

One question which could not be answered by the measurements is how this would look in a more resource constrained environment where the nodes for example are PDA's - they have much less CPU power than 2Ghz Pentium 4 CPU's and power consumption is much more of an issue than with notebook computers. Therefore the use of a netfilter match which reduces the amount of work could actually make a difference.

7.1 Future Development

One possible extension could be the avoidance of passing packets to userspace: A generic interface like Netlink could be used to send information to the kernel for publishing and to receive requests from incoming packets. This way the complete packet modifications could be done inside the kernel and userspace would not have to be concerned about the exact packet format nor handling of actual packets. But since the measurements with the current SLP implementation show that the performance does not suffer significantly from passing around the packets, the performance gain from this might be negligible.

Something which would be very useful is a generic testbed for testing and measuring the performance of the implementation. This could be something like the APE (Ad hoc Protocol Evaluation) testbed[1] but with a stronger adaptation to the given environment. Some of the scripts developed in this thesis might be used as a start, but they still lack a lot of genericity.

[1] `http://apetestbed.sourceforge.net/`, last updated in 2002, so probably not really active anymore

List of Figures

Listings

List of Tables

Bibliography

[1] Manuel Graber. Distributed service discovery and session initiation using sip for manets. Master's thesis, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2005.

[2] IETF Network Working Group. Service location protocol, version 2, 1999. RFC 2608, `http://www.ietf.org/rfc/rfc2608.txt`.

[3] IETF Network Working Group. Ad hoc on-demand distance vector (aodv) routing, 2003. RFC 3561, `http://www.ietf.org/rfc/rfc3561.txt`.

[4] Patrice Oehen and Martin Kos. Distributed service location protocol for mobile ad hoc networks, 2006. Labor Thesis.

[5] Rusty Russell, Harald Welte, and mailing list `netfilter@lists.samba.org`. Linux netfilter hacking howto, 2002. `http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html`.

[6] Rusty Russell and mailing list `netfilter@lists.samba.org`. Linux 2.4 packet filtering howto, 2002. `http://www.netfilter.org/documentation/HOWTO//packet-filtering-HOWTO.html`.

[7] Patrick Stuedi, Manuel Graber, and Gustavo Alonso. Exploiting synergies between service discovery and routing in wireless multihop networks. Technical report, Swiss Federal Institute of Technology Zurich (ETHZ), Department of Computer Science, 2006.

[8] Uppsala University. Ad-hoc on-demand distance vector routing - for real world and simulation. `http://core.it.uu.se/AdHoc/AodvUUImpl`.

[9] David A. Wheeler. Program library howto, 2003. `http://www.tldp.org/HOWTO/Program-Library-HOWTO/`.