

Implementation of HEAT for Linux 2.6 with IPv6

Semester Thesis

Sebastian Altorfer

26.07.2007

Advisor: Rainer Baumann
Supervisor: Prof. Dr. Bernhard Platter
Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

HEAT is a scalable and robust routing protocol for wireless mesh networks. In this thesis, we present an IPv4 and IPv6 implementation of HEAT for Linux 2.6. It is partially implemented in the kernel and user space, and can be run in mesh node and gateway operation mode. We validate our implementation on a small testbed in different static as well as dynamic scenarios. In addition, we explain how to set up an IPv6-to-IPv4 transport relay translator and DNS proxy.

Contents

1	Introduction	5
2	Related Work	6
2.1	Field-based Routing	6
2.2	Implementation of AODV-UU in IPv6	7
3	Design	8
3.1	Overview	8
3.2	User Space	9
3.2.1	Timers	9
3.2.2	Routing	10
3.3	Kernel Space	10
3.3.1	Host module	10
3.3.2	Gateway module	11
4	Implementation	13
4.1	Environment	13
4.2	Code	13
4.3	Data structures	14
4.3.1	Hello Packet	14
4.3.2	Source Routing Header	14
4.3.3	Routing Table	15
4.3.4	Binary Tree	16
4.4	Retrieving IPv6 addresses in user space	16
4.5	Definitions	17
4.5.1	Packet Types	17
4.5.2	Time Intervals	17
4.6	Netfilter Hooks	17
4.6.1	Host Module Hooks	17
4.6.2	Gateway Module Hooks	18
5	IPv6-to-IPv4 Transport Relay Translator	19
5.1	Setting up a TRT to work with HEAT	20
6	Validation	22
6.1	Route Recording and Source Routing	22
6.2	Dynamic Neighbor Selection	22
6.2.1	Dynamic neighbor selection based on incoming hello messages	22
6.2.2	Dynamic neighbor selection based on aging	23
6.3	A Walk through the WMN	23
7	Conclusion	25
8	Future Work	26
9	Bibliography	27

A	Instructions for running HEAT and setting up a TRT	28
A.1	How to install the HEAT protocol implementation	28
A.1.1	Note for Knoppix users:	28
A.1.2	How to activate the IPv6 mode:	29
A.2	How to configure the network devices	29
A.2.1	Configuration for the IPv6 case:	29
A.2.2	Note for the IPv6 case:	30
A.3	How to run the program	30
A.4	How to clean up	30
A.5	How to see some kernel debugging output	31
A.5.1	How to see additional debugging output in the IPv6 case:	31
A.6	How to configure the IPv6-to-IPv4 TRT in the IPv6 case	31
A.6.1	How to install and set-up the totd DNS proxy:	32
A.6.2	How to install and set-up pTRTd:	32
A.6.3	General IPv6-to-IPv4 environment:	32
A.6.4	Exemplary configurations of eth0, eth1 and eth2:	33
A.6.5	Configuration of the resolv.conf file for hosts in the WMN:	33
A.6.6	Testing the TRT:	33

1 Introduction

Today, hosts that want to participate in the Internet require a connection to an internet service provider or an other entity that offers the same functionality. There exists a wide range of possible ways to establish such a connection. One of these possibilities is to use a wireless connection, such that the host maintains wireless communication with a counterpart which itself is connected to the Internet. The mentioned counterpart is often called an access point. This way of connecting to the Internet is getting more and more popular, since wireless communication between computer hosts is widely standardized and available. However, with today's standards and technology, an access point can only connect a limited number of hosts to the Internet and these hosts have to be within a certain range of the access point. For a large number of hosts even in a relatively small area, for example in an urban environment, a large number of pccess points is required to provide Internet connectivity for all those hosts. Another drawback is the limited mobility of hosts, because the wireless LAN (IEEE 802.11-family) standard does not support handover.

But with an ever increasing number of hosts capable of wireless communication, a new possibility of connecting them to the Internet arises. If the density of hosts that can use wireless connections (hereby referred to as nodes) is high enough, this constellation of hosts and the access points in between is a hybrid Wireless Mesh Network (WMN). Hybrid WMNs are networks that interconnect mobile and fixed hosts with access points connected to the Internet. If every node acts as a host and as a relay for the communication in the WMN at the same time, then there is no need for every node in the WMN to be directly connected to an access point, it suffices that it can reach at least another node which in turn is able to reach other parts of the WMN and at some point an access point (gateway) to the Internet.

A similar mode of operation is already available with today's standards. In the ad-hoc configuration every node is able to reach every other node inside the network. But ad-hoc networks operate without access points and they are only used for communication within a set of nodes.

If one wishes to provide Internet connectivity for all nodes in an mobile ad-hoc network one must solve several problems. First of all, there is the question of how to assign each node in the network an unique IP address. This problem is actually solved when using IPv6 and its autoconfigurative properties[1]. The IPv6 stateless address autoconfiguration assigns every interface an unique link-local address. An additional problem is the absence of any kind of hierarchy in a mobile ad-hoc network. Thus, simply adding Gateways to the network wouldn't accomplish anything since the nodes have no way of learning about these gateways. Furthermore nodes may freely join and leave the mesh network at any time and at any rate. This leads to a fast and ever changing topology of the network.

It is evident that even if one can solve the problems mentioned above there still remains the question of how to actually perform any kind of routing in such WMNs.

If the gateways are known to the nodes inside the network one must provide some way of reliably routing outgoing traffic towards these gateways and back to the nodes. Commonly used routing protocols such as Distance Vector Routing (DVR) or Link State

routing (LS) fail because of the dynamic property of wireless hybrid mesh networks. The main problem being scalability.

The Computer Engineering and Networks Laboratory at ETH Zurich proposed a new approach to obtain a scalable routing protocol for a unknown number of hosts in a hybrid wireless mesh network. This approach is a Field-based Routing (FBR)[2] protocol that relies on a temperature field to route data packets called HEAT [3]. HEAT is based on assigning each participant of a network a field intensity to establish a scalar field. Using HEAT each node dynamically and continuously calculates one of their neighbors to be their default gateway for all traffic towards the Internet. The path back from the gateway to the node is determined via source routing. The paths are stored at the gateway. The path is previously recorded hop by hop in every packet that traverses the network towards the gateway and is extracted and saved upon reaching it. Through the use of this approach the desired scalable and hierarchic routing protocol can be realized.

The goal of this thesis is to implement HEAT for Linux 2.6 such that it can be used either with IPv4 or IPv6. There already exists an IPv4 implementation of HEAT for Linux 2.6 which we used as a starting point for an implementation which is also usable with IPv6.

The previous implementation of HEAT used an implementation of another routing protocol for ad hoc networks called AODV-UU [4] as a reference which provided some insight into how routing protocols are generally implemented in Linux. We also wanted our implementation not to be dependent on additional third party libraries that offer generalized network and packet handling functionalities for both IPv4 and IPv6. The implementation of HEAT requires two basic mechanisms. One to distribute every node's field intensity and IP address to all it's neighbors and to calculate the dynamic field. For this purpose we used a HELLO mechanism similar to the one in the AODV-UU implementation. Additionally there is the need to handle data packages for recording the path of the packet to the gateway. This is done through the use of netfilter hooks that filter the in- and outgoing packages in a kernel module. By the design of HEAT every node only needs to add one additional entry in the kernel routing table and thus the requested scalability is granted.

2 Related Work

In this section we will explain shortly HEAT and introduce the IPv6 capable implementations of AODV-UU that we examined.

2.1 Field-based Routing

The basic idea of HEAT is to view all the nodes in a WMN as entities that have a field intensity. The network then forms a kind of scalar field. A temperature field is such a scalar field and is of special interest to us. In physics one can calculate the propagation of heat by collisions between particles by a discrete solution of the Poisson equation. This solution has one interesting property concerning our routing problem. The intensity of the resulting field decreases away from the heat source thus resulting in a gradient. In this analogy we view nodes as particles and gateways as heat sources.

While the node’s field intensities are continuously calculated every node simply determines it’s neighbor with the highest field intensity as the standard gateway and sends all outgoing packets to it. This of course requires a field construction algorithm that prevents local maxima, leading to a monotonic field.

The information needed by every node to construct a field with the above properties, namely the field intensities of neighboring nodes and their neighbors are exchanged between the nodes with HELLO messages. An aging mechanism ensures that a failing node no longer influences the field. The field calculation function calculates a field intensity based on the information provided by the HELLO messages. Let x_i denote the field intensities of a node’s neighbors. The following Algorithm is the field calculation function proposed by the Computer Engineering and Networks Laboratory at ETH Zurich:

Algorithm 1 Field Calculation Function

```

1 : Sort  $x_i$  in ascending order
2 :  $l = 0$ 
3 :  $f(0) = 0$ 
4 : while  $f(l) < x_i$  do
5 :    $f(l + 1) = f(l) + (x_i - f(l)) \cdot \kappa$ 
6 :    $l = l + 1$ 
7 :   Go to next node in sorted list
8 : end while
9 :  $f_{final} = f(l)$ 

```

Using this algorithm nodes in areas with a high node density and thus a high redundancy are assigned a higher field intensity and therefore a higher priority. This effect is controlled by the parameter κ and gets stronger with a decreasing κ . A node only uses the field intensities of neighbors that did not themselves use the nodes field intensity in their field intensity calculation. It has been proven that this field calculation function avoids local maxima and that it can handle node departures and arrivals [2, chapter II, section B].

2.2 Implementation of AODV-UU in IPv6

AODV-UU is a Linux-based implementation of the Ad-hoc On-demand Distance Vector Routing (AODV) developed at the Uppsala University. We examined the IPv6 patch for AODV-UU version 0.9 by Martin Dietze at the University of Buckingham. What made it interesting for us was the fact that the original code is for IPv4 and that the IPv6 capable implementation is based on the original code. We too had an implementation of HEAT for IPv4 and based our IPv6 capable implementation upon it. To make the code compilable for either IPv4 or IPv6 a layer for IP version abstraction called *iplib* is added. Within the AODV-UU code all types and calls to the Linux kernel or socket interfaces specific to the IP version are replaced by calls to the respective *iplib* types and functions [5]. As an alternative we also examined the work of Peter Lee at Simon Fraser University [6]. It is based on AODV-UU version 0.5 and does not use *iplib*. It provided some insight how the port of a networking application to IPv6 is done but for some key issues, i.e. obtaining the IPv6 addresses of a given interface from user space this code also did not provide any satisfying solution.

3 Design

In this section we describe the design of our HEAT implementation. Many design principles are the same for the IPv4 and the IPv6 case. There are however some key differences that we will specifically address.

3.1 Overview

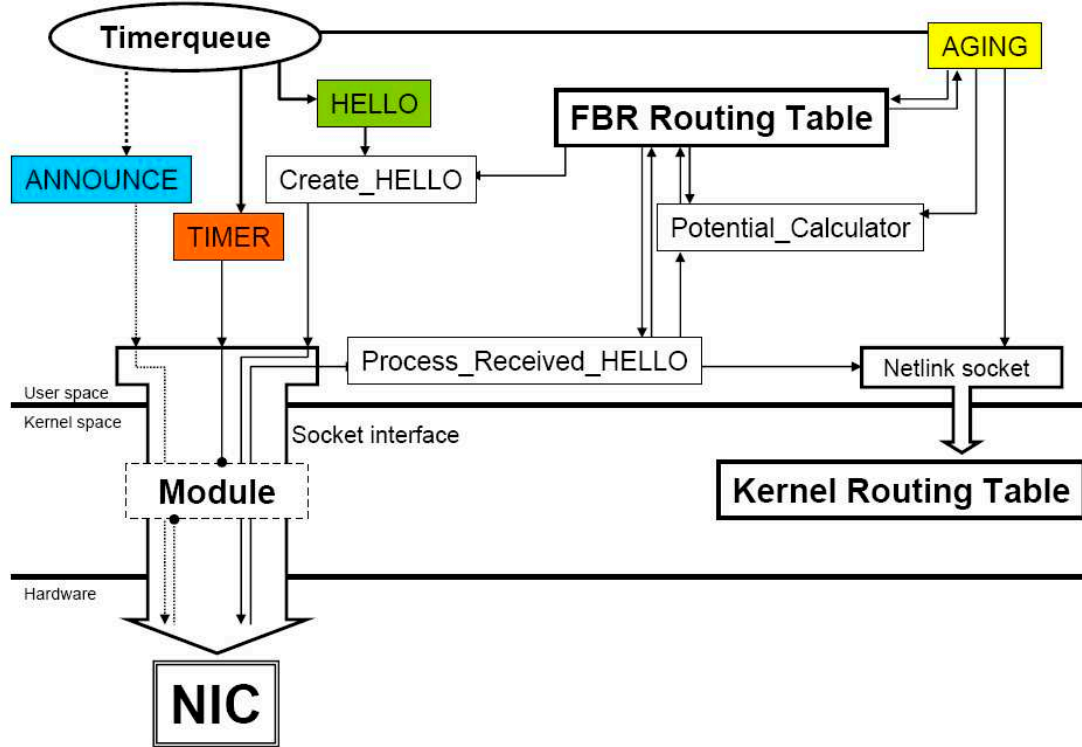


Figure 1: HEAT implementation, user space overview

The code of our implementation of HEAT is divided into two parts. One part is the code for the user space daemon and the other part is the code for the two kernel space modules. The daemon process handles the sending of hello messages, the calculation of the field intensity and it determines and writes the default gateway in the kernel routing table.

In the kernel space there is one kernel module for the case where our implementation is run on a normal host and one for the case where it is run on a gateway. The kernel modules handle the packet filtering and the insertion, modification and removal of a special header in the packets. The kernel module for the gateway configuration also manages and stores the recorded routes to nodes in the WMN. The recorded routes are used for source routing packets back from the gateway to the nodes in the WMN.

3.2 User Space

The implementation of the user space daemon has two main parts. One part is the handling of all the required timers that clock the application. The other part handles the routing by determining the default gateway (a default gateway in this context is the default route for all outgoing packets in the kernel routing table) and writing it into the kernel routing table.

Both these two parts require additional data and functions in the user space:

- The **FBR Routing Table** is an internal routing table and contains all neighbors of a node and their field intensities, sorted by their intensities.
- The **Intensity_Calculator** calculates a node's field intensity using **Algorithm 1** and the information in the **FBR Routing Table**. This function is only used if the application is run in the host mode. In the gateway mode the node has a fixed field intensity.
- The Netlink socket is used to communicate with the kernel to write or delete entries in the **Kernel Routing Table**. In the IPv6 case the Netlink socket is also used to obtain the IPv6 Addresses of network devices.
- The **Socket Interface** is used in the entire application to send and receive packets.

3.2.1 Timers

The timers are controlled by the **Timerqueue**, the central unit in the user space (see Fig. 1 on page 8). The four timers are as follows:

HELLO: The hello timer calls **Create_HELLO** in fixed intervals to create the hello messages that contain the sending node's field intensity and it's neighbors that contributed to it's field intensity in the first place. This additional information is available through the **FBR Routing Table**. This information is included for the receiver of a hello message to perform poison reverse. The actual hello message is broadcast (multicast in the IPv6 case) as a UDP packet via the socket interface. Incoming hello messages pass the loaded kernel module and are processed in the usersapce daemon by **Process_Received_Hello** and the node's intensity is recalculated via **Intensity_Calculator**. If a received hello message comes form a node with a high enough field intensity it is set as the new default gateway in the **Kernel Routing Table** via the Netlink socket interface.

AGING: This timer periodically ages the entries of the **FBR Routing Table**. If an entry is not updated in a predefined time it is deleted from the table. This is done to ensure that a neighboring node that either failed, changed it's field intensity or left the range of the node to which the **FBR Routing Table** belongs does not influence the field intensity. This also applies for the default gateway, if the node no longer receives hello messages from it's default gateway and it is therefore deleted by aging a new default gateway is set.

TIMER: This timer is only active in the gateway mode. The recorded routes from nodes inside the WMN to the gateway are stored at the gateway. These paths are not stored indefinitely and they are aged using this timer. It sends special timer messages that are caught by the gateway module that calls **FBR GW Binary Tree**, a data structure in the gateway kernel module to delete the obsolete outdated paths.

ANNOUNCE: This timer is only active if the application is run with the corresponding feature enabled. It is used by nodes to periodically send an announce message to the gateway to prevent the source routing path from being deleted at the gateway. This might be necessary in cases of nearly one way traffic from the Internet to a node inside the WMN, i.e. streaming applications.

3.2.2 Routing

This part of the user space implementation is only used if we are in the host mode. Every time a host receives a hello message it recalculates its field intensity using the **Intensity Calculator** function and updates the **FBR Routing Table**. If the neighbor with the highest field intensity has changed it will be the new default gateway for the host and all future packets will be routed to that node. The old default gateway is removed and the new default route is added to the **Kernel Routing Table** by sending requests for the kernel to the Netlink socket.

3.3 Kernel Space

The kernel space part of our implementation is strictly divided into two separate parts. There is a module for the host mode and one for the gateway mode. Every module has its separate code.

The host module handles the traffic inside the WMN. It is responsible for recording the routes from nodes inside the WMN to a gateway and for the actual source routing. The gateway module extracts the recorded routes from the packets that come from inside the WMN towards the Internet.

Addressing: As a speciality of the IPv6 case the gateway module also has to take care of a replacement for Network Address Translation (NAT) to handle the addressing of the packets at the gateway. The hello messages in the WMN are sent, received and addressed using link-local addresses that are automatically generated using the MAC number of the network device. However if one wishes to send and receive packets to and from the Internet one needs an address with a global scope. One way to handle this problem is to assign the hosts in the WMN global scope addresses in a separate subnet or one can integrate a mobility management protocol at the gateways [7]. In the IPv4 case this is solved using NAT.

3.3.1 Host module

In the host mode a node is participating in the source routing by recording the route to the gateway and forwarding packets in the WMN to their destinations. To accomplish

that we need to catch and filter all packets that arrive at the node. This is done in the host kernel module. Nodes inside a WMN receive hello and data packages. Hello messages pass through the host module and are processed by `Process_Received_HELLO` in the user space daemon.

Source Routing: To record the routes from nodes inside the WMN to a gateway the host module inserts a header (encapsulation) into all locally generated packets. This additional header contains the recorded route for the source routing. This FBR header is inserted between the network and the transport layer for the IPv4 case. In the IPv6 case it is inserted directly after the 40 byte long IP header. When a packet traverses a node on it's way to the gateway this node just adds it's own address to the FBR header of the packet.

If a packet that is on it's way from the gateway to a node inside the WMN traverses a node it extracts the address of the next hop from the source routing header that is included in the packet and sets this address as the new destination of the packet. If a node is the final destination of a packet the FBR header is removed (decapsulation) and the packet is passed on.

Packet Filtering: To catch the packets that arrive at the host node we use two Netfilter hooks, `PREROUTING` and `POSTROUTING` (see Fig. 2 on page 12). These hooks catch the packets that are sent from and to the host and they process the packets before passing them on. Packets that are broad- or multicast (only multicast in the IPv6 case) always pass through the hooks unmodified.

- If a node is the destination of a data packet it is caught by the `PREROUTING` hook of the node's host module and the FBR header is removed by `Decapsulate`. Packets that are addressed to a different node in the WMN are also caught and the address of the next hop is set in the IP header by `Update` and the packet is forwarded.
- If a node receives a data packet on it's way to the gateway it is caught by the `POSTROUTING` hook and the node's address is added to the FBR header by `Add` to record to route of the packet through the WMN. If a node creates packets or announce messages they are caught by the same hook and the host module inserts a FBR header into the packet by `Encapsulate`. Locally generated hello messages pass through the hooks unmodified.
- As a speciality of the IPv6 case the host module has to let some ICMPv6 packets go through the Netfilter hooks unchanged. Namely ICMPv6 neighbor solicitation and advertisement packets have to be sent between hosts inside the WMN. Since HEAT does not support direct traffic between hosts inside the WMN these ICMPv6 packets associated with the neighbor discovery protocol for resolving IPv6 addresses to link-layer addresses must not be encapsulated.

3.3.2 Gateway module

A node running in the gateway mode is on the border between the WMN and the Internet. The gateway module has to remove the FBR header (decapsulation) from outgoing

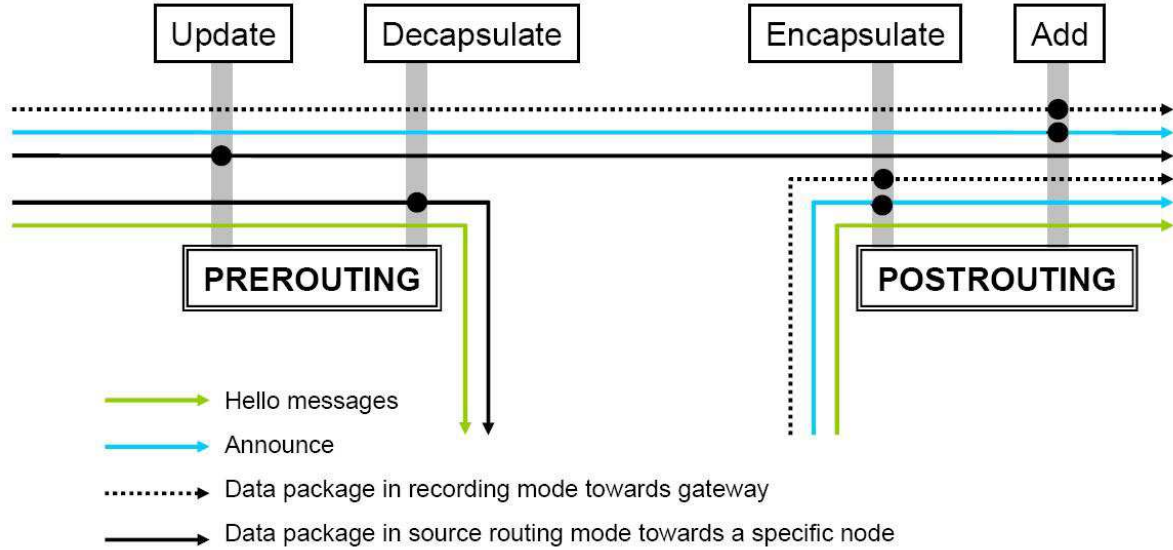


Figure 2: HEAT implementation, host kernel module packet filtering

packets to the Internet. For the source routing it also has to insert the FBR header (encapsulation) into packets coming from the Internet with a destination inside the WMN. This header contains the route through the WMN to the destination. The recorded routes are stored in the **FBR GW Binary Tree**.

In the IPv6 case where there is no NAT the gateway module also has to take care of the addressing of the incoming packets if the hosts inside the WMN do not use global scope addresses in their own subnet.

We use Netfilter hooks to catch and filter incoming packets at the gateway that are not being broad- or multicast in the gateway module.

Packet Filtering: In the gateway module there are the three Netfilter hooks **PREROUTING 1**, **PREROUTING 2** and **LOCAL_OUT** (see Fig. 3 on page 13. In the IPv4 case the NAT takes place between **PREROUTING 1** and **PREROUTING 2**.

- Incoming announce messages at the gateway are caught by the **PREROUTING 1** hook and dropped and the **Decapsulate** routine refreshes the announced route in the **FBR GW Binary Tree**. Data packages to the Internet are first caught in **PREROUTING 1** where the FBR header is removed from the packets by **Decapsulate**. The removal of the FBR header has very high priority and has to happen before NAT in the IPv4 case.
- The data packets from the Internet to a destination inside the WMN are caught by the **PREROUTING 2** hook because it has to pass through NAT first in the IPv4 case. In the IPv6 case the destination addresses of the packets are changed by **Encapsulate** which also includes the FBR header into the packets.
- The **LOCAL_OUT** hook catches the locally in the user space generated timer messages. Whenever this packet is caught it is dropped and the entries of the **FBR GW Binary Tree** are aged by **Aging**.

Incoming and locally generated HELLO messages pass through the hooks unaltered.

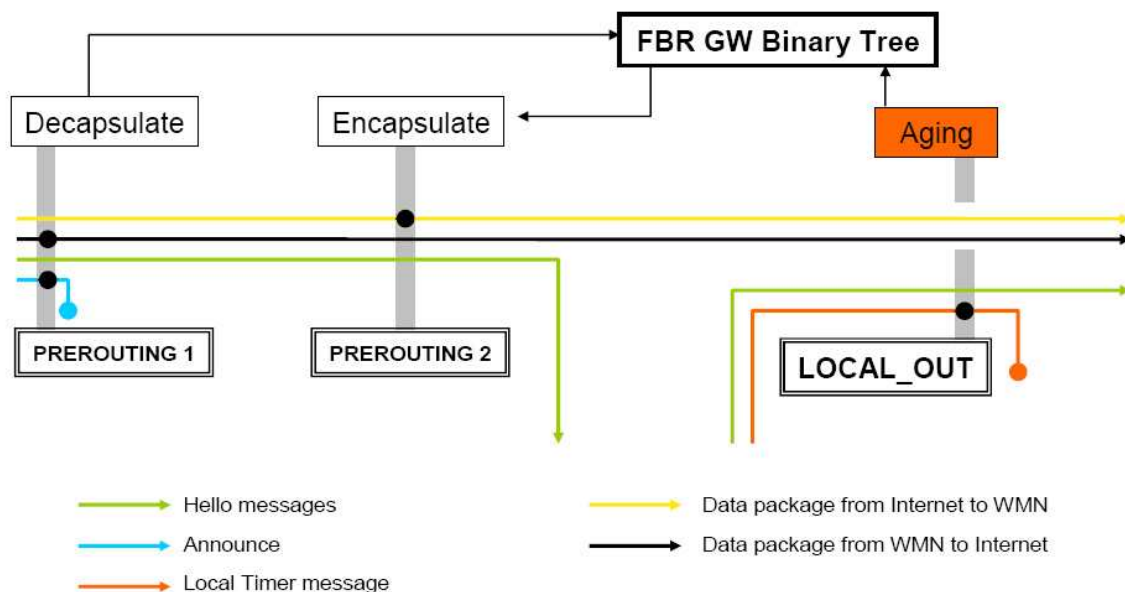


Figure 3: HEAT implementation, gateway kernel module packet filtering

4 Implementation

In this section we focus on specific details of our implementation of HEAT. We first present the used environment and then do some general remarks about the code. Further we will describe the important data structures and functions of our implementation.

4.1 Environment

We developed our implementation of HEAT on IBM T42 laptops equipped with an Atheros (AR52512) wireless network interface cards. The used operating system was Knoppix v5 with Kernel 2.6.17. We compiled the code with gcc 4.0.4 using libc 2.3.6.

4.2 Code

The code of our implementation is based on the IPv4 implementation of HEAT [8], that used the AODV-UU implementation as a reference. The timerqueue in particular and the timer handling were reused without major modifications in both implementations of HEAT.

Since our implementation should be usable with IPv4 and IPv6 we wrote all the code for both cases into the same files. The distinction between the two cases is done using precompiler statements:

```
#ifdef IPV6
#else
#endif
```

Writing the code this way it suffices to use an additional option for the `make` command to enable or disable the IPv6 version of our implementation.

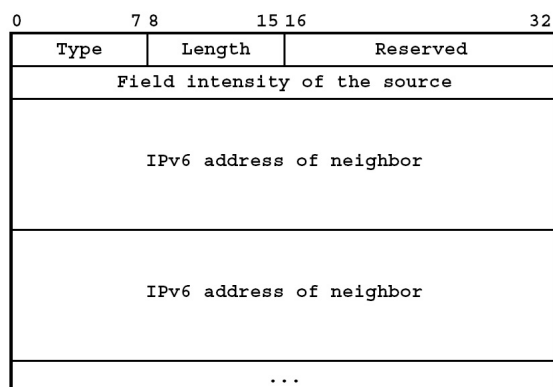
The IPv4 implementation of HEAT was very well documented and we too documented our code in a way that future work based on our code is easily realizable. All the following code samples are for the IPv6 case only.

4.3 Data structures

4.3.1 Hello Packet

```
#define FBR_MAX_NEIGHBOR 15

typedef struct {
    u_int32_t type:8;
    u_int32_t len:8;
    u_int32_t reserved:16;
    u_int32_t field;
    struct in6_addr ips[FBR_MAX_NEIGHBOR];
} fbr_hello_msg;
```

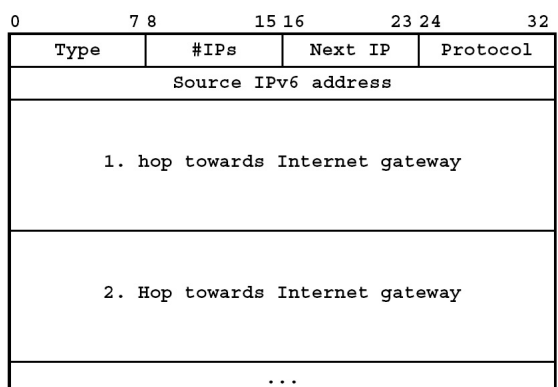


The hello packet is used by the nodes to multicast their field intensity and the list of contributing neighbors to the surrounding nodes. The field intensity is needed by the neighboring nodes to calculate their own intensity. Together with the list of contributing nodes every node makes sure that it does not use the field intensity of a node that calculated its own intensity using this node's intensity in its calculations. This is similar to poison reverse in classical distance vector routing and is implemented to avoid a swingdown of field intensities as a consequence of a node failure or loss of connectivity. Currently the hello packets all have the same size, but the hello packet already has a length field for a future modification to handle dynamic hello packet sizes. The hello packet is also used to send the timer and announce messages.

4.3.2 Source Routing Header

```
#define FBR_MAX_ROUTE 10

struct fbr_sr_hdr {
    u_int8_t type:8;
    u_int8_t nr_ips:8;
    u_int8_t next_ip:8;
    u_int8_t nexthdr:8;
    struct in6_addr ips[FBR_MAX_ROUTE];
};
```



The source routing struct is inserted as the FBR header into every packet that is transmitted in the WMN. It is directly inserted after the IPv6 header of the packets. This header contains all the information that the gateway needs for source routing packets from

the Internet back to a node inside the WMN. It also contains the information needed by the nodes for the source routing inside the WMN. If the type field is set to `FBR_SR_RECORD` it designates a packet that is on it's way to the gateway and every node that it passes writes it's own IP address into the source routing header at the position `next_ip` of the `ips` array. If the type is set to `FBR_SR_FORWARD` we are dealing with a packet that is coming from the gateway towards a node in the WMN. Every traversed node then sets the new destination address of the packet by reading the IP address stored at `next_ip` in the `ips` array.

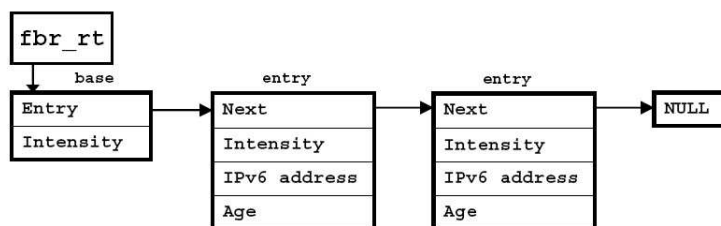
The `nexthdr` field specifies the (extension) header that originally followed the IPv6 header. In the original IPv6 header the `nexthdr` field is set to `IPPROTO_FBR` which we set to `0xc8` because it is not currently assigned to any (extension) header. This way the Netfilter hooks can easily detect if a packet is encapsulated or not. Upon decapsulating the `nexthdr` field in the IPv6 header is set to the value in `nexthdr` again. The `FBR_MAX_ROUTE` constant specifies how many IPs for source routing can be stored in the source routing header. With this static size of the source routing header it is easier for the nodes that process the encapsulated packets.

The inserting of the source routing header after the IPv6 header actually against protocol since some extension headers are required to follow immediately after the IPv6 header or to appear in a certain order. Every device has to parse the extension headers in the exact order that they arrive in. Simply inserting our source routing header after the last extension header would be an option. This however requires that there exist functions to parse the IP extension headers reliably. This was not yet the case with the kernel that we used to used to develop this implementation.

4.3.3 Routing Table

```
typedef struct fbr_rt_e {
    struct fbr_rt_e *next;
    u_int32_t field;
    struct in6_addr id;
    u_int8_t age;
} fbr_rt_entry;
```

```
typedef struct {
    fbr_rt_entry *entry;
    u_int32_t field;
} fbr_rt_base;
```



The routing table has two parts, the base and a linked list of entries that are ordered by their field intensities. Since the number of neighbors of a node in a WMN will remain relatively small a linked list is efficient enough. The base of a node's routing table contains the field intensity of the node itself and it points to the first entry of the linked list. Each entry contains the field intensity of the node that is associated with the IP address in that entry. If a node in that linked list used the node that owns this linked list to calculate it's field intensity the field intensity of that entry is set to zero. Every entry also contains a byte long `age` field. If the `age` byte of an entry is set to zero by the aging of the routing table that entry is deleted. Every entry also contains a `next` field, a pointer to the next

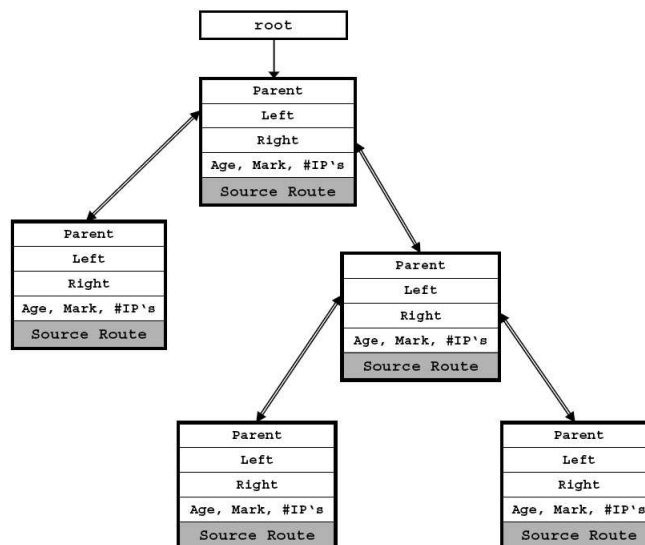
entry in the linked list. A global pointer to the base of the routing table exists in the user space daemon.

4.3.4 Binary Tree

```
#define FBR_MAX_ROUTE 10

struct fbr_bt_node {
    struct fbr_bt_node *parent;
    struct fbr_bt_node *left;
    struct fbr_bt_node *right;
    int age;
    int mark;
    int nr_ips;
    struct in6_addr ips[FBR_MAX_R(
};

struct fbr_bt_node *root
```



The binary tree is part of the gateway kernel module. It is used to save the recorded routes for the source routing back to the node in the WMN. The routes are ordered by their destination addresses inside the WMN. Every entry of the binary tree stores the route, the length and the age of the route. Additionally it stores the administrative information that is needed by the functions that handle the binary tree. Every entry has pointers to every direction and a mark that indicates if the entry has already been visited by some algorithms. We chose to store the routes in a binary tree since there may be a large number of routes to store at the gateway that are frequently accessed. In this situation a linked list is not efficient enough. Other possibilities are a hash table or a balanced binary tree.

4.4 Retrieving IPv6 addresses in user space

This task might seem rather simple and basic as it is easy to obtain the IPv4 address of a network interface using a `ioctl` systemcall with the `SIOCGIFADDR` option. For the IPv6 case however one can't use this systemcall. One way to obtain the IPv6 addresses of a network interface is to read the local configuration files of said network interface. We chose a different approach and used the Netlink socket interface to send a special request for the IP addresses of the network devices of the system to the kernel. We then process the answers that the kernel sends back to get the desired IP address. The request is basically set up the following way:

```
struct {
    struct nlmsg_hdr nl_hdr;
    struct ifaddrmsg ifaddr_msg;
} fbr_nl_msg;
```



```
fbr_nl_msg.nl_hdr.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifaddrmsg));
fbr_nl_msg.nl_hdr.nlmsg_type = RTM_GETADDR;
fbr_nl_msg.nl_hdr.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
fbr_nl_msg.nl_hdr.nlmsg_pid = getpid();
fbr_nl_msg.ifaddr_msg.ifa_family = AF_INET6;
```

4.5 Definitions

4.5.1 Packet Types

Our implementation of HEAT uses six packet types. The `FBR_TYPE_ANOUNCE` is used to announce routes to the gateways. The `FBR_TYPE_TIMER` type is used to age the binary tree in the kernel space part of the application. The `FBR_TYPE_LOCAL` type is currently unused. It may be used for a future work that enables traffic inside the WMN.

```
#define FBR_TYPE_HELLO 0
#define FBR_TYPE_ANOUNCE 1
#define FBR_TYPE_SR_RECORD 2
#define FBR_TYPE_SR_FORWARD 3
#define FBR_TYPE_LOCAL 4
#define FBR_TYPE_TIMER 5
```

4.5.2 Time Intervals

Every timer has its own interval. The jitter interval specifies the time frame for blurring the time interval between two consecutive hello packets to prevent repetitive collisions.

```
#define FBR_HELLO_INTERVAL 1000
#define FBR_ANOUNCE_INTERVAL 3000
#define FBR_BT_INTERVAL 1000
#define FBR_AGE_INTERVAL 1000
#define FBR_JITTER_INTERVAL 100
```

4.6 Netfilter Hooks

The two kernel modules each use their own set of Netfilter hooks to catch and process packets.

4.6.1 Host Module Hooks

The host module has a pre routing and a post routing hook for catching packets. The pre routing hook is used to catch packets that are being source routed through the WMN and packets that are destined for the host itself. This hook has high priority. The post

routing hook catches packets that are on their way to a gateway and or packets that were locally generated by the host. This hook as normal priority.

- **pre routing hook:**

```
static struct nf_hook_ops fbr_input_hook = {
    .hook = fbr_input_filter,
    .owner = THIS_MODULE,
    .pf = AF_INET6,
    .hooknum = NF_IP6_PRE_ROUTING,
    .priority = NF_IP6_PRI_FIRST,
};
```
- **post routing hook:**

```
static struct nf_hook_ops fbr_output_hook = {
    .hook = fbr_output_filter,
    .owner = THIS_MODULE,
    .pf = AF_INET6,
    .hooknum = NF_IP6_POST_ROUTING,
    .priority = NF_IP6_PRI_FILTER,
};
```

4.6.2 Gateway Module Hooks

The gateway module has three hooks. Two pre routing hooks with different priorities and one local out hook. The pre routing hook with high priority catches packets coming from inside the WMN. In the IPv4 case NAT takes place in between the two pre routing hooks. The normal priority pre routing hooks then (after NAT in the IPv4 case) catches packets coming from the Internet. The local out hook catches the local timer messages that are being sent from the user space daemon of the gateway.

- **pre routing hook 1:**

```
static struct nf_hook_ops fbr_input_high_hook = {
    .hook = fbr_input_high_filter,
    .owner = THIS_MODULE,
    .pf = AF_INET6,
    .hooknum = NF_IP6_PRE_ROUTING,
    .priority = NF_IP6_PRI_FIRST,
};
```
- **pre routing hook 2:**

```
static struct nf_hook_ops fbr_input_low_hook = {
    .hook = fbr_input_low_filter,
    .owner = THIS_MODULE,
    .pf = AF_INET6,
    .hooknum = NF_IP6_PRE_ROUTING,
    .priority = NF_IP6_PRI_FILTER,
};
```

- **local out hook:**

```
static struct nf_hook_ops fbr_loc_out_hook = {
    .hook = fbr_loc_out_filter,
    .owner = THIS_MODULE,
    .pf = AF_INET6,
    .hooknum = NF_IP6_LOCAL_OUT,
    .priority = NF_IP6_PRI_FILTER,
};
```

5 IPv6-to-IPv4 Transport Relay Translator

For validation purposes we investigated and employed an IPv6-to-IPv4 Transport Relay Translator (TRT)[9]. The purpose of a TRT is for hosts in an IPv6-network to be able to gain access to IPv4-only network resources, for example IPv4-only web servers.

We investigated a TRT for connections initiated by IPv6-only and destined for IPv4-only hosts (see Fig. 4 on page 20), because TRT is designed to require no additional modifications on IPv6-only initiating hosts nor on IPv4-only destination hosts. Routing inside the IPv6 network has to be configured so that packets to `fec0:0:0:ffff::/64` are routed toward the TRT. The subnet `fec0:0:0:ffff::/64` does not exist inside the IPv6 network. When the initiating host (IPv6 address `A6`) wants to send a packet to the destination host (IPv4 address `X4`) it has to send the packet to `fec0:0:0:ffff::X4`. The packet is then routed to the TRT and captured by it.

In the case of a TCP connection the TRT accepts the TCP/IPv6 connection between the initiating host `A6` and `fec0:0:0:ffff::X4` and communicates with `A6` using TCP/IPv6. Then the TRT takes the last 32 bits of the address `fec0:0:0:ffff::X4` to get the real IPv4 destination `X4` of the TCP connection. The TRT then makes a TCP/IPv4 connection from `Y4` to `X4` and forwards the traffic across the two TCP connections. For this task we used the portable TRT daemon (pTRTd) [10].

Address mapping: An initiating IPv6 host must use a special form of IPv6 address (addresses with the dummy prefix) to connect to a IPv4 destination host. The initiating host needs to be able resolve the address of an IPv4 host in this special form. We used a Domain Name System (DNS) proxy for this task, the trick or treat daemon (totd)[11]. The DNS proxy is accessible via IPv6 inside the IPv6 network and accepts the DNS queries for an IPv4 host sent by the initiating host `A6` and forwards them to either an IPv6 or IPv4 DNS server. The totd receives the DNS reply from the DNS server and it then sends a new DNS reply to the initiating host containing an AAAA entry (an IPv6 address) composed of the dummy prefix and the IPv4 address that was sent in the original DNS reply.

The hosts inside the IPv6 network only have to be configured to send their host name resolution queries to the DNS proxy, the DNS proxy behaves like a real DNS server to the hosts sending queries.

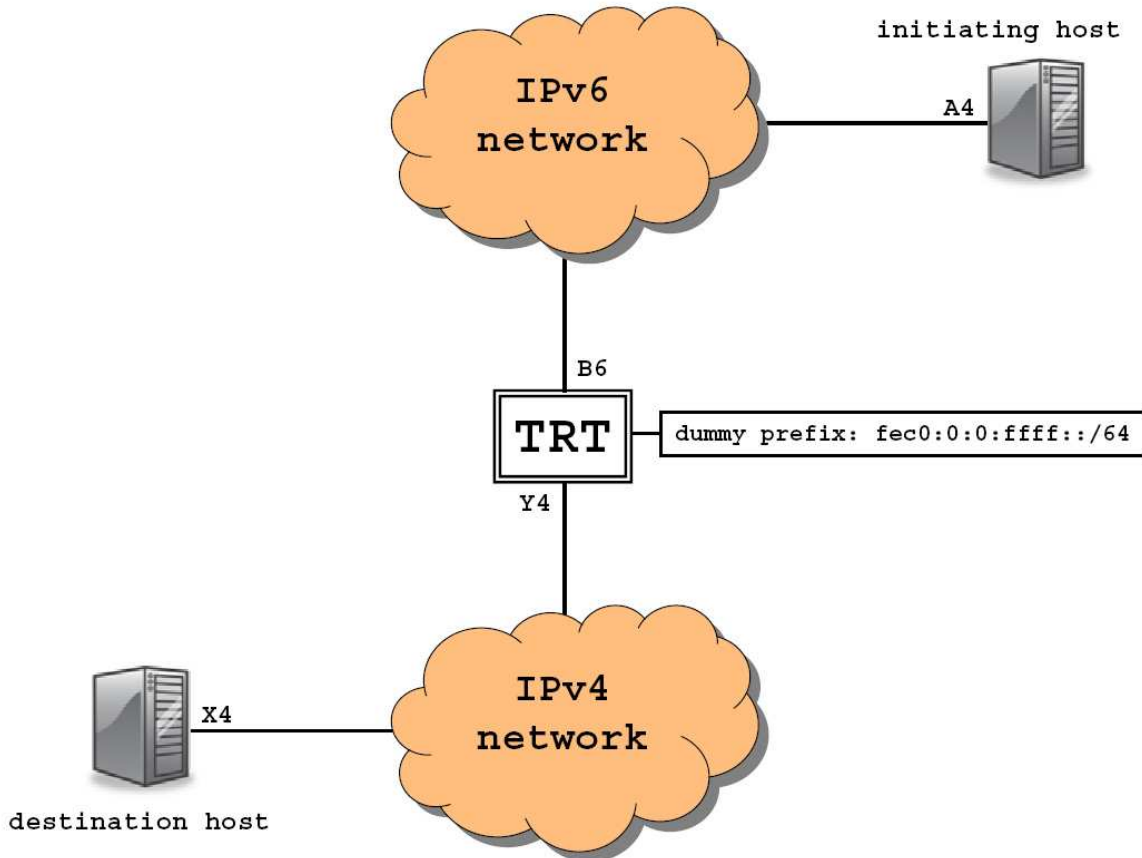


Figure 4: IPv6-to-IPv4 Transport Relay Translator set-up

5.1 Setting up a TRT to work with HEAT

It was our goal to use TRT to enable hosts inside a WMN running our implementation of HEAT to be able to access IPv4 hosts in the Internet (see Fig. 5 on page 21).

Addressing and Routing: Our implementation of HEAT on the hosts inside the WMN continuously recalculates its default gateway and writes it into the kernel routing table as the default route. Our implementation relies on the assumption that all outgoing traffic of a host for a destination outside of the WMN uses the default route to work properly. However, this default route is only used if the destination is not in the same subnet with the sending host. Our implementation does not support the direct communication between hosts inside the WMN and it is therefore necessary that all resources that hosts inside the WMN want to access are located outside the WMN and that they are in a different subnet than the hosts inside the WMN.

For this reason the hosts inside the WMN (including B6, the IPv6 address of the wireless device of the HEAT gateway) are in one subnet, and the link that connects the HEAT gateway and the TRT is in a different subnet (IPv6 addresses C6 and D6).

Configuration of the hosts inside the WMN and the HEAT gateway: The hosts inside the WMN only have to be configured to use the DNS proxy for the resolution of domain names. The HEAT gateway has to be configured to forward all traffic destined for the non-existent subnet given by the dummy prefix toward the TRT. This is done by adding a default route to the kernel routing table of the HEAT gateway and thus can only work properly if the subnet given by the dummy prefix does not exist on the site.

Configuration of the TRT: On the TRT two applications have to be configured, pTRTd and totd. They both have to be configured to use the same dummy prefix and the totd has to forward the DNS queries it receives to the IPv4 DNS server with the address Z4 (see appendix for more detailed instructions).

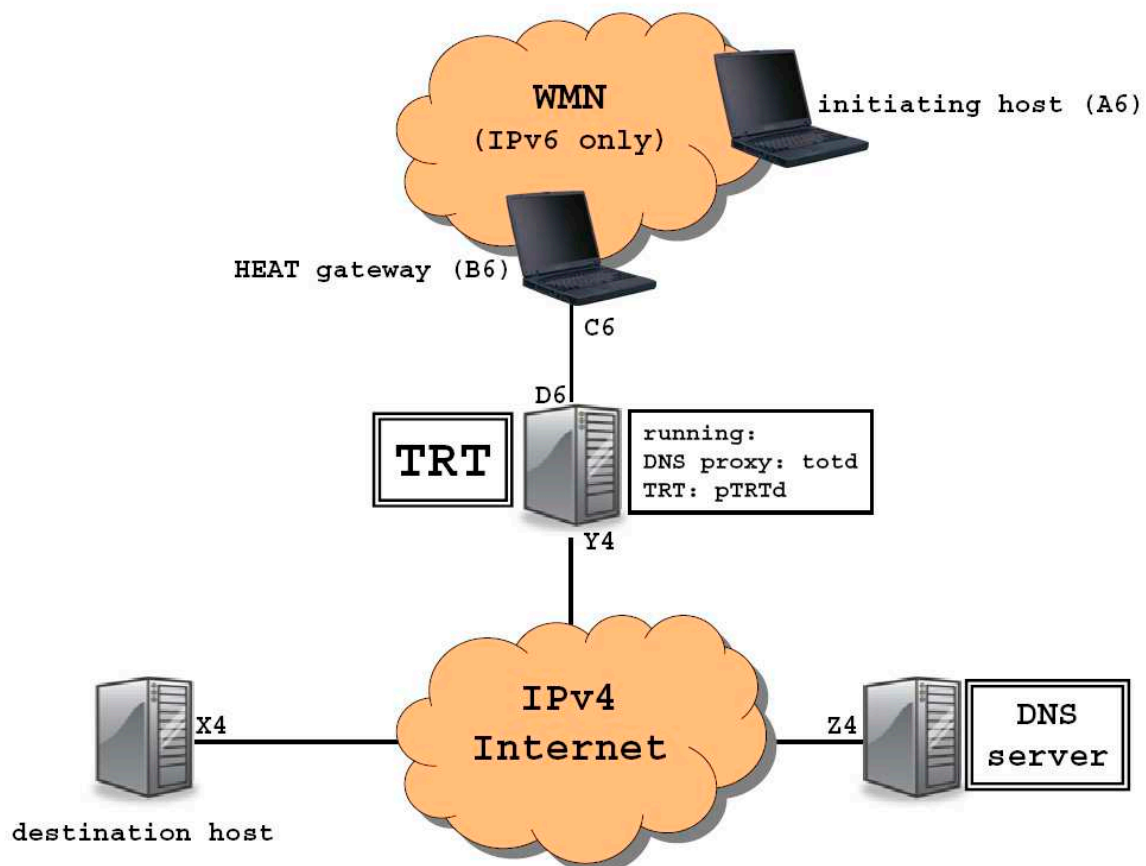


Figure 5: IPv6-to-Ipv4 TRT and a WMN

6 Validation

To validate our implementation of HEAT for IPv6 we set up four different environments. We validated the following functionalities:

- Route recording and source routing
- Dynamic neighbor selection
- Dynamic gateway selection

The validation took place in the ETZ building at ETH Zurich. In every set up the HEAT gateway was connected to the IPv4 Internet using a TRT (see Fig. 5 on page 21). We pinged IPv4 hosts such as `www.google.com` and we used the debugging options of our implementation (see appendix on how to enable them) for validating the different functionalities. We used the same set ups for the validation that were used to evaluate the IPv4 implementation of HEAT.

6.1 Route Recording and Source Routing

To validate the route recording and source routing functionalities we arranged a HEAT gateway and 4 nodes in a chain (see Fig. 6 on page 22). If the user on the host U pings an IPv4 host the data packets traverse along the nodes 3, 2, 1 and finally the gateway G and vice versa. On the packet's way to the IPv4 host the path along the nodes is recorded and this stored path is then used for source routing the answer packet back to the node U. To ensure that the data packets are encapsulated and decapsulated properly we used a packet sniffer (Ethereal [12]) at every node.



Figure 6: HEAT gateway and 4 nodes inside the WMN

6.2 Dynamic Neighbor Selection

6.2.1 Dynamic neighbor selection based on incoming hello messages

To validate that our implementation of HEAT also works with two gateways we set up two HEAT gateways and two nodes in between (see Fig. 7 on page 23). We ran pings to an IPv4 host on the mobile host U. It's starting position was A where it sent all it's packets to the node 1, the only node in range at that position. As we moved towards the position C we entered the range of the gateway on the right side of our set up at position B. Since the right gateway has a higher field intensity than the node 1 the host U immediately changed it's default route to the right HEAT gateway upon receiving a hello message from this gateway containing the higher field intensity value. We observed this behavior by monitoring the kernel routing table of the host U by running the command: `watch route -A inet6`.

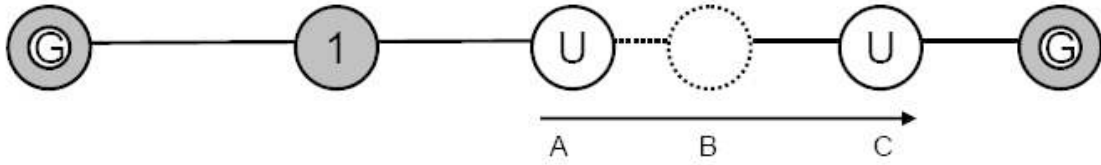


Figure 7: Two HEAT gateways and two nodes inside the WMN

6.2.2 Dynamic neighbor selection based on aging

To ensure that defective gateways and nodes are not used for routing inside the WMN we set up two HEAT gateways and two nodes in between the gateways (see Fig. 7 on page 23). At time A the host U sends all its packets to the gateway on the right side. At the time B we turned off the right gateway (see Fig. 8 on page 23). At time C after not receiving any new hello messages from the right gateway in 3 seconds, a time interval in which a functioning node sends up to three hello messages, the right gateway is removed from U's neighbor table and it chooses its neighbor with the next highest field intensity value, node 1 as its new default route. The host U is still sending packets to the right gateway during the 3 second timeout interval and packet loss occurs.

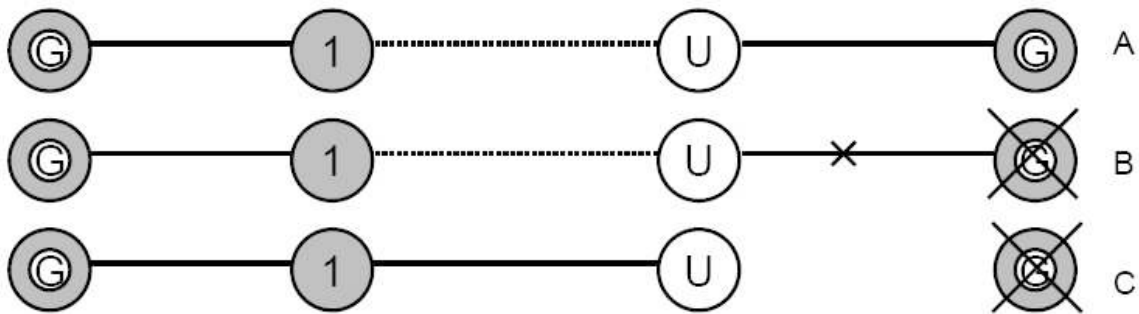


Figure 8: Turning off one of two HEAT gateways with two nodes inside the WMN

6.3 A Walk through the WMN

As a final step of our validation we wanted to test if a mobile host inside the WMN does always get connectivity to the Internet and if the dynamic neighbor selection is working properly in a more complex set up (see Fig. 9 on page 24).

- (A) At the time A the node 1 is the only node inside the WMN in range of the gateway. The nodes 2 and 3 are both in range of node 1 but not in range of each other. At the time A the mobile host U only has the node 2 in range and it uses it as its default route.
- (B) At the time B the mobile host U is in range of both node 2 and node 3. It still uses node 2 as its default route since nodes 2 and 3 have the same field intensity values.

- (C) The host U now uses the node 3 as it's default route. Some packet loss occurs because this change happened due to aging of the neighbor table. The node 2 was removed as the default route for U 3 seconds after U received the last hello message from node 2.
- (D) The mobile host U still uses node 3 as it's default route, node 3 is the only node in range of U.
- (E) U immediately started using 1 as it's default route upon receiving the first hello message from the node 1 which has a higher field intensity than node 3.

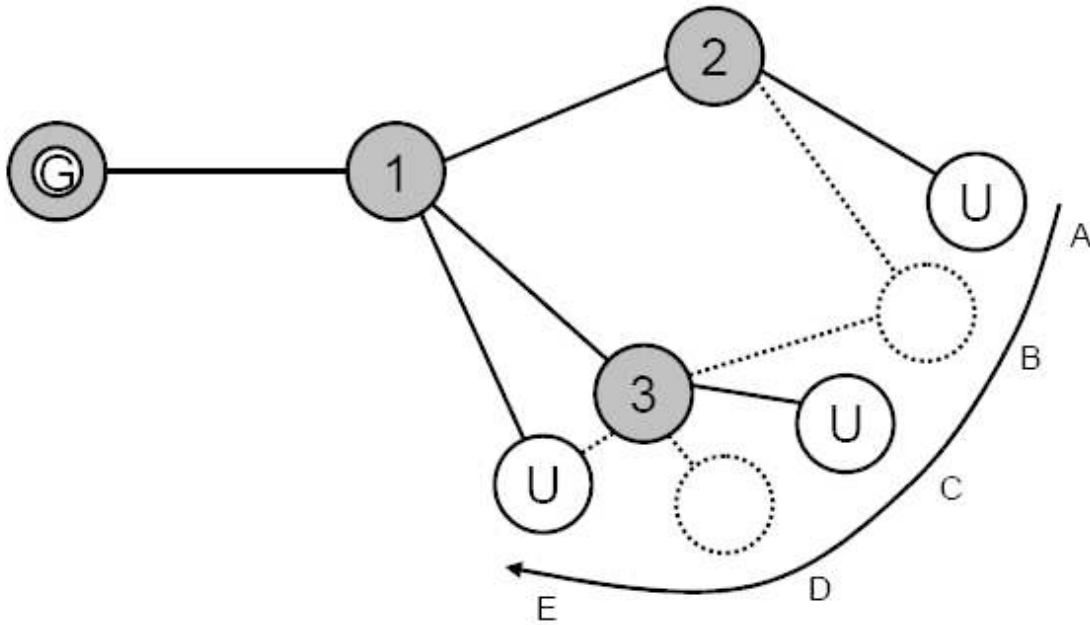


Figure 9: One HEAT gateway and three static nodes and one mobile host inside the WMN

7 Conclusion

The Communication Systems Group of the Computer Engineering and Networks Laboratory at ETH Zurich developed HEAT, an approach for a scalable field-base routing protocol for Wireless Mesh Networks. An IPv4 implementation of HEAT for Linux 2.6 for IPv4 was developed at ETH Zurich at the end of last year. Our contribution is the IPv6 implementation of HEAT on Linux 2.6. In addition we investigate and set up an IPv6-to-IPv4 Transport Relay Translator to provide access to the IPv4 Internet for hosts inside an IPv6-only WMN.

We investigate several implementations of AODV-UU for IPv6 and the existing IPv4 implementation of HEAT. We further analyze the kernel code of the IPv6 implementation for Linux 2.6. We decided to base our implementation directly on the IPv4 implementation of HEAT and our code is compilable for the IPv4 as well as the IPv6 case.

Our implementation consists of three parts: The user space daemon, the host kernel module and the gateway kernel module. The user space daemon mainly handles the field calculation and the neighbor table. The host kernel module's main task is the route recording for packets on their way towards a gateway and source routing of packets that are destined for a host inside the WMN. The gateway module's main task is to extract the recorded routes from packets that are destined for a host outside the WMN and store them and to insert the appropriate route for source routing into packets that are destined for hosts inside the WMN.

We elaborate comprehensive instructions how to set up and configure a TRT to work with our implementation of HEAT. Using the TRT we validate that the different functionalities of our implementation worked according to the design of HEAT.

Our implementation of HEAT is available as open source software.

8 Future Work

In this section we propose what in our opinion should be the next steps in further developing the implementation of HEAT.

- **Integration of a Mobility Management Protocol at the HEAT gateways:**

The implementation of such a protocol would allow multi-homing and switching access networks for nodes inside the WMN.

- **Adjusting the transmit power for hello packets:**

During our validation we observed that hello packets tend to have a higher range than normal data packets. This difference was significant enough to sometimes cause total packet loss when hellos were still being received but data packages were lost between nodes. We thus propose to send hello packets at a slightly reduced power level.

- **Dynamic field intensities at the gateways for load balancing:**

If there are multiple gateways in a WMN this would be a way to enable a gateway with a high load not to be the default route for traffic toward the Internet anymore by reducing it's own field intensity.

- **Dynamic length of the source routing header:**

In the current implementation the source routing header has a fixed size, however most of the time not all the reserved fields are used. By making the size of the source routing header dynamic one could reduce the overhead that is being introduced by our implementation of HEAT. In our implementation we already added a length field to the source routing header for this purpose.

- **Extensive evaluation in a large testbed:**

We only validated that our implementation is working properly with a maximum of 5 used nodes. To actually evaluate the performance of our implementation one needs to use more nodes in a larger environment.

9 Bibliography

References

- [1] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 2462 (Draft Standard), December 1998.
- [2] Rainer Baumann, Simon Heimlicher, Martin May, Vincent Lenders, Karoly Farkas, and Bernhard Plattner. *Field Based Interconnection of Hybrid Wireless Mesh Networks*. Computer Engineering and Networks Laboratory, ETH Zurich, 2006.
- [3] Rainer Baumann, Vincent Lenders, Simon Heimlicher, and Martin May. *HEAT: Scalable Routing in Wireless Mesh Networks Using Temperature Fields*. Computer Engineering and Networks Laboratory, ETH Zurich, 2006.
- [4] Erik Nordstroem. Ad hoc On-Demand Distance Vector Routing. <http://core.it.uu.se/core/index.php/AODV-UU>, December 2006.
- [5] Martin Dietze. An IPv6-Patch for AODV-UU 0.9. <http://webpace.buckingham.ac.uk/cadams/bnrg/aodv6/index.html>.
- [6] Peter Lee. AODV For IPv6 Network. <http://members.shaw.ca/aodv6-sfu/>.
- [7] Rainer Baumann, Olga Bondareva, Simon Heimlicher, Vincent Lenders, and Martin May. *A Macro Mobility Notification Protocol for Hybrid Wireless Mesh Networks*. Computer Engineering and Networks Laboratory, ETH Zurich, 2007.
- [8] Michael Steiger and Sascha Trifunovic. *Implementation of a Field Based Routing Protocol for Wireless Mesh Networks*. Computer Engineering and Networks Laboratory, ETH Zurich, 2006.
- [9] J. Hagino and K. Yamamoto. An IPv6-to-IPv4 Transport Relay Translator. RFC 3142 (Informational), June 2001.
- [10] Nathan Lutchansky. Portable Transport Relay Translator Daemon. <http://www.litech.org/ptrtd/>.
- [11] Feike W. Dillema. The totd (Trick Or Treat Daemon) DNS proxy. <http://www.vermicelli.pasta.cs.uit.no/software/totd.html>.
- [12] Gerald Combs. Ethereal: A Network Protocol Analyzer. <http://www.ethereal.com/>.

A Instructions for running HEAT and setting up a TRT

A.1 How to install the HEAT protocol implementation

First of all the source code needs to be compiled. There is a Makefile which does all the work for. However the code consist of two parts, the daemon and the modules. The modules, unlike the daemon, which is compiled against the libc headers, are compiled against the kernel headers. In kernel 2.6 you will need a precompiled kernel to be able to compile any modules. To make things even easier, just follow the step by step instructions: Change to the folder where the kernel sources belong:

```
# cd /usr/src
```

If not there, download them. Use apt-get on Debian or Knoppix. Replace x with your actual version:

```
# apt-get update
# apt-get install linux-source-2.6.x
```

Now there should be a file in the current folder with the name `linux-source-2.6.x.tar.bz2`. Decompress it:

```
# tar -xjf linux-source-2.6.x
```

Give the folder a nicer name:

```
# mv linux-source-2.6.x linux-2.6.x
# cd linux-2.6.x
```

Copy the current config file:

```
# cp /boot/config-2.6.x .config
```

Compile:

```
# make
# make modules
```

A.1.1 Note for Knoppix users:

For some Knoppix versions there are no kernel sources available via apt-get. In this case download the DVD image of Knoppix and copy the kernel sources from there. Now everything is ready to compile the HEAT implementation:

```
# cd /path to the fbr source
```

A.1.2 How to activate the IPv6 mode:

```
# make IPVERSION=-DIPV6
# make install
```

For the IPv4 case use:

```
# make
# make install
```

A.2 How to configure the network devices

To be able to run the program the network devices have to be configured first. Every computer needs at least a wireless card; the gateway needs an other device, connected to the internet. In the following text we assume the wireless device is called `ath0`. In order to be able to build an ad-hoc network, all the hosts need to be part of the same subnet, share the same Service Set Identifier (SSID), be part of the same cell and communicate over the same channel.

The easiest way to configure the device, especially if you need it more than once is to add the following lines into `/etc/network/interfaces`:

```
iface ath0 inet static
    address 192.168.1.1
    netmask 255.255.255.0
    broadcast 255.255.255.255
    wireless-mode ad-hoc
    wireless-channel 1
    wireless-essid test
```

A.2.1 Configuration for the IPv6 case:

The following lines have to be added to the `/etc/network/interfaces` file for the IPv6 case:

```
iface ath0 inet6 static
    address 2001::61
    netmask 16
    wireless-mode ad-hoc
    wireless-channel 1
    wireless-essid test
```

Then start the device:

```
# ifup ath0
```

A.2.2 Note for the IPv6 case:

If the `ifup ath0` command does not work properly there is another way to configure the network device. Add only the following lines to the `/etc/network/interfaces` file:

```
iface ath0 inet6 static
    address 2001::61
    netmask 16
```

Then run the following commands:

```
# wlanconfig ath0 destroy
# wlanconfig ath0 create wlandev wifi0 wlanmode ad-hoc
# iwconfig ath0 channel 1
# iwconfig ath0 essid test
# ifup ath0
```

The cell cannot be configured automatically, so do it now:

```
# iwconfig ath0 ap 01:02:03:04:05:06
```

Obviously the address, netmask, cell or any other parameter can be changed to correspond your needs.

A.3 How to run the program

After performing all the steps explained above you can start the program. Assuming `eth0` is the device connected to the Internet.

In host mode:

```
# fbr -i ath0
```

In gateway mode:

```
# fbr - i ath0 -g eth0
```

If you want to run it in the background just add the option `-d`.

If you need help:

```
# fbr -h
```

If it still doesn't work, mail us.

A.4 How to clean up

To stop the program use `ctrl+c` if it is visible or kill the process if invisible. This is done the following way:

Get the process id:

```
# ps -e
```

Kill the process

```
# kill id
```

To uninstall the whole thing:

```
# cd /path to the fbr source  
# make uninstall
```

And clean up:

```
# make clean
```

A.5 How to see some kernel debugging output

The daemon informs you about the important changes of topology and the current potential and gateway. This information cannot be turned off; you can only demonize (detach from terminal) the process.

In the modules any output is disabled and can only be enabled at compiling time. To do so uncomment the `#define FBR_DEBUG` statement in `fbr-mod.h` of both modules. This enables a bunch of `printk`.

The output of the `printk` is not visible in a normal terminal. If you want to see them, you can e.g. change the screen with `ctrl+alt+F1` and you will be flooded with information about every packet coming in and going out and the binary tree and so on... filter it yourself.

A.5.1 How to see additional debugging output in the IPv6 case:

For additional debugging messages add `#define FBR_DEBUG_SPAM` and `#define FBR_DEBUG_SPAM_2` to the files `defs.h`, `mod_gw/fbr-mod.h` and `mod_host/fbr-mod.h`.

A.6 How to configure the IPv6-to-IPv4 TRT in the IPv6 case

For the TRT you will need a separate machine with two network devices running on a Linux (we used Knoppix) that supports dual-stacking and you need a connection to the IPv4 Internet.

You will need the following third party applications:

- **totd, Trick or Treat Daemon** DNS proxy, <http://www.vermicelli.pasta.cs.uit.no/software/totd.html>
- **pTRTd, Portable Transport Relay Translator Daemon**, <http://v6web.litech.org/ptrtd/>

A.6.1 How to install and set-up the totd DNS proxy:

First, download the source code and decompress it (suppose 1.4 is the release number):

```
# tar xvzf totd-latest.tar.gz
# cd totd-1.4
# ./configure
# make depend
# make
# make install
```

Then copy the file `totd.sample.conf` from the directory you have de-compressed the totd source code (`totd.tgz`) to `/usr/local/etc/totd.conf`:

```
# cp totd.sample.conf /usr/local/etc/totd.conf
```

Then edit the configuration file for your needs. `man totd` for more information about the individual options. It is important that you set the prefix in the configuration file to `fec0:0:0:ffff::/64`. This is the default prefix that pTRTd uses.

Then you can start the totd with

```
# totd
```

A.6.2 How to install and set-up pTRTd:

For pTRTd to work it is required that the binary `sbin/ip` from the `iproute-packet` is available. It is also necessary that the `tun/tap-driver` (<http://vtun.sourceforge.net/tun/>) is present in the kernel.

After decompressing the source simply run:

```
# ./configure
# make
# make install
```

You can now run pTRTd:

```
# ptrtd
```

A.6.3 General IPv6-to-IPv4 environment:

The TRT is connected to the HEAT gateway and to the IPv4 Internet (see. Fig. 4 on page 20).

Let's assume that the device that connects the HEAT gateway to the TRT is `eth0`, the device at the other end of that connection at the TRT is `eth1` and the device at the TRT that connects it to the IPv4 Internet is `eth2`

A.6.4 Exemplary configurations of eth0, eth1 and eth2:

Add the following lines to the `/etc/network/interfaces` files of the corresponding machines:

- `eth0`:

```
iface eth0 inet6 static
    address 2002::61
    netmask 16
    gateway 2002::62
```
- `eth1`:

```
iface eth1 inet6 static
    address 2002::62
    netmask 16
```
- `eth2`:

```
auto eth2
iface eth2 inet dhcp
```

It is crucial that the hosts in the WMN and the link that connects the TRT and the FBR gateway are in different subnets!

A.6.5 Configuration of the `resolv.conf` file for hosts in the WMN:

The `/etc/resolv.conf` file of hosts inside the WMN should only contain one entry:

```
nameserver 2002::62
```

Where `2002::62` is the address of the TRT which may be different in your configuration.

A.6.6 Testing the TRT:

The following tests are to be performed with HEAT running on a host inside the WMN. You can test if the TRT is working by first pinging the TRT:

```
# ping6 2002::62
```

where `2002::62` is the address of the TRT in our case.

If this works you can then do a DNS lookup of an IPv4 host:

```
# dig www.google.com ANY
```

You should get an answer with some AAAA entries that have the IPv6 prefix that you previously configured.

Then you can test if the pTRTd is working by pinging an IPv4 host:

```
# ping6 www.google.com
```

Note that if you get answers it is the pTRTd that is answering, meaning that it is running. For a final test you can then use the IPv6 capable browser of your choice (we used Firefox) to access an IPv4 website. It should load and display as if you were on a regular IPv4 host connected to the IPv4 Internet.