Implementation of a Field Based Routing Protocol for Wireless Mesh Networks

Semester Thesis

Sascha Trifunovic
Michael Steiger

December 21, 2006

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

Abstract

Field Based Routing is a scalable routing protocol for Wireless Mesh Networks. In this thesis we implemented a Field Based Routing Protocol on Linux 2.6. We validated and evaluated it in static and dynamic scenarious. The evaluation showed that FBR works and behaves as dynamically as requested.

Contents

1 Introduction

At the beginning of the Internet age every computer was directly or indirectly cable-connected to the Internet. With upcoming wireless technology, the demand of Internet access to be available everywhere has risen and is now a matter of course. Within an area around a wireless router or access point connectivity is granted. Among other reasons, due to the limited range of wireless antennas many access points are needed, each requiring an Internet connection of its own. Most of these connections are only partially used and thus most of the time idle, which is an enormous waste of bandwidth.

Given areas where the density of WLAN-enabled hosts i.e. mobile devices is high enough, there is a possibility to get connected to the Internet in a new way. In the constellation of some access points and many mobile devices we get a mesh network - a hybrid Wireless Mesh Network (WMN) [1]. If, besides its interest to get connected to the Internet, every participating device is acting as relay for other devices too, there is no need for a mobile user to have a direct wireless connection to an access point. It is sufficient to have connectivity to another mobile device in the WMN who is directly connected or also (in a recursive manner) connected indirectly. Thus every node is a user and supporting pillar of the network.

Unlike in common networks, where network parameters like IP address, subnet mask et cetera were automatically configured via DHCP, in Mobile Ad Hoc Networks (MANET) [2] there is no such type of service. Mobile devices have to be manually set to a static IP address. This leads to the first problem: There is no hierarchy and thus no information about the location of the gateways. Another problem is the dynamic state of hybrid wireless mesh networks. Mobile devices frequently join and leave the network or change their location. The topology varies fast.

In addition we have a problem with scalability. Common routing protocols like Distance Vector Routing (DVR) [3] or Link State Routing (LSR) [4] fail in scalability due to fast changing and mobile participants. Link breaks could not be handled fast enough. If every device has to store a route to every participant, routing tables would become extremely huge and furthermore a standard device is not optimized for large routing tables. To reach a scalable routing protocol with unknown number of hosts and solving the address hierarchy problem a new approach called field based routing (FBR) [5] was established by the Computer Engineering and Networks Laboraty at ETH Zurich.

FBR is based on potentials i.e. field intensities as base for a one way hierarchy. The idea of FBR is that mobile devices chose one of their neighbours as default gateway where all the packages are sent to. The backward path is found via source routing i.e. recorded route. Using this approach a hierarchic and scalable routing protocol is realizable [5].

Our task is to implement such a protocol in Linux. There already exist some protocols for ad-hoc networks like e.g. Ad Hoc On Demand Distance Vector Routing (AODV) [6]. AODV is a reactive routing protocol which means that

a route to another device is created on demand and thus it is relatively scalable [7]. Our interest in AODV was to see how routing protocols are generally implemented. Despite AODV is based on totally different idea, some approaches like the integrated HELLO mechanism or the timer handling have convinced us to build up FBR on AODV.

FBR needs two basic mechanisms. To distribute the potential and IP address to a device's neighbours and to build the dynamic field, a HELLO mechanism is used. On the other side there is the data package handling. This is done by a kernel module whose netfilter hooks filter the package flow. Our approach achieves that every device has only one routing table entry. Unlike in AODV where the routing information to deliver a package is stored in routing tables, Source Routing accomplishes that all routing information is part of the package itself. Scalability is granted.

2 Related Work

In this chapter we shortly introduce Field Based Routing FBR and related routing protocol implementations like Ad-Hoc On Deman Distance Vector Routing (AODV) and Dynamic Source Routing (DSR).

2.1 Field Based Routing FBR

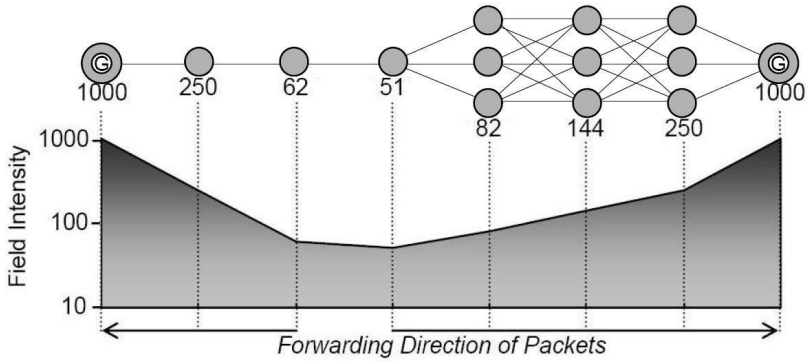


Figure 1: Example of potential topology

The idea of FBR is based on potential or temperature fields. The object is to build a potential field as it is shown in Figure 1. If a node wants to send a data package to the Internet it just takes the neighbour node with the highest potential as its default gateway where all packages are sent to. This procedure guarantees that an Internet gateway is always reached. E.g. for the node with potential 62 in Figure 1 the default gateway is the node to the left with potential 250.

As the path to the Internet uses the steepest gradient, we do not have any hierarchic rule for the backward path. Hence source routing is applied using the previously recorded routes. Due to the one way hierarchy there is no other way to get back. To build a potential field as shown in Figure 1 the Computer Engineering and Networks Laboratory at ETH Zurich provided Algorithm 1.

Algorithm 1

Sort potentials x_i in descending order $l = 0$
$f(0) = 0$
while $f(l) < x_i$
$f(l+1) = f(l) + (x_i - f(l)) \cdot \kappa$
$l = l + 1$
end while
Potential $= f(l)$

For this algorithm to work a hello mechanism is needed. This hello mechanism is implemented as following: In predefined intervals every node broadcasts

its address and potential to all neighbours. The Internet gateways always emits the highest potential. When starting the procedure common meshnodes have a potential of 0. Every time a node receives such a hello message its own potential is recalculated using algorithm 1. In the next hello message a node broadcasts, the new potential is included. In [5] loopfreeness of networks which are built on algorithm 1 is showed.

2.2 Implementation of other routing protocols

To get some impressions how routing protocols for ad-hoc networks are generally implemented we studied several approaches. Our goal was to design FBR for kernel 2.6. Finally we implemented FBR based on AODV-UU.

2.2.1 AODV-UU

Uppsala University implemented an Ad Hoc On Demand Distance Vector Routing protocol for Linux kernel 2.4 and 2.6. We evaluated version 0.9.3. AODV-UU [8] works in ad-hoc networks and creates on request a route to another node. For that several types of messages like route request (RREQ), route reply (RREP) and route error (RRER) are used. When a route is not used or needed anymore it will time out and gets deleted. AODV-UU works with a user space daemon and additional modules for the package handling. We successfully ran AODV-UU.

2.2.2 kernel-aodv

kernel-aodv [9] was designed for kernel 2.4 by the National Institute of Standards and Technology (NIST). It is totally implemented in kernel space as a module. We tried to run it, but after two days of debugging we decided to concentrate on AODV-UU. In addition, a huge effort would had been necessary to change a module for kernel 2.4 to kernel 2.6 without guarantee if it would acutally work.

2.2.3 DSR-UU

In DSR-UU [10] source routing is implemented. A virtual network interface is created so that DSR can coexist with a standart ad-hoc network. In addition a protocol handler is implemented. We dealed with DSR-UU only a short time and decided to implement source routing in a diffrent way.

3 Design

3.1 Overview

The FBR Routing Protocol is based on four timer driven mechanisms in the user space, an intern routing table called *FBR Routing Table*, a netlink socket, a socket interface and two kernel modules (one for host and one for gateway implementation). The gateway implementation contains a *FBR GW Binary Tree* to store recorded routes to nodes in the WMN.

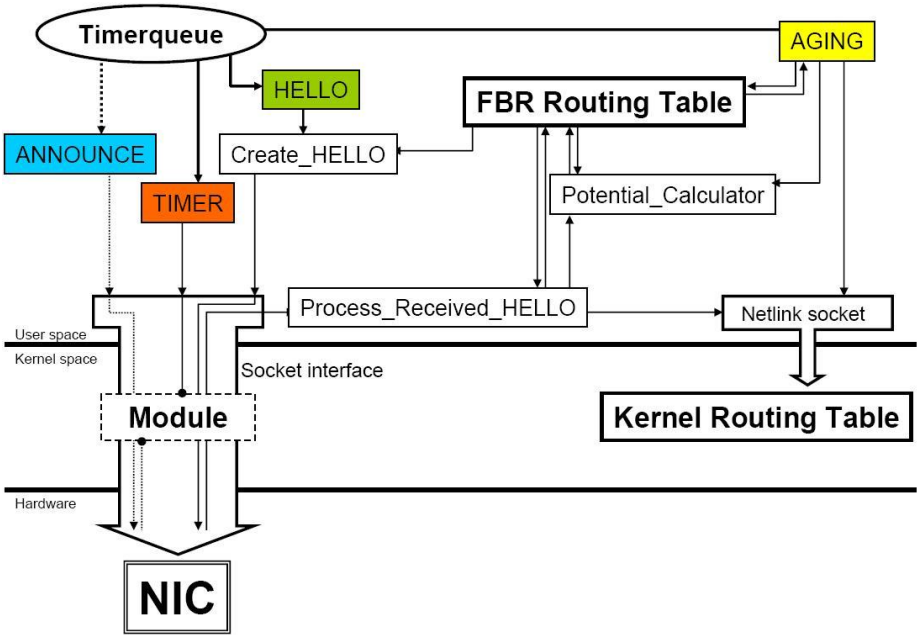


Figure 2: User Space Overview Field Based Routing

3.2 User Space

The *Timerqueue* ist the central unit in user space. It controls *HELLO* (green), *AGING* (yellow), *ANNOUNCE* (blue) and *TIMER* (orange). These four processes are periodically executed in predefined time intervals. The *FBR Routing Table* contains information about all neighbours and their potentials. The *Potential_Calculator* recalculates the current potential of the node - in gateway mode it is not executed. To write and actualize the default gateway of a node a netlink socket to communicate with the *Kernel Routing Table* is used.

HELLO The hello process for broadcasting a node's potential is always active when FBR is running. Periodically *Create_HELLO* is called and the necessary information like current potential and IP addresses of nodes which were used to calculate the actual potential (to implement a poison reverse mechanism) is included from *FBR Routing Table*. Via the socket interface passing through the module the UDP hello message is broadcasted.

Incoming hello messages, after they are let through by the module are caught by *Process_Received_HELLO*. The *FBR Routing Table* is accessed and the potential is recalculated via the *Potential_Calculator*. If due to changes a new default gateway has to be set, the *Kernel Routing Table* is actualized via the netlink socket.

AGING The aging process manages the *FBR Routing Table*. An entry in *FBR Routing Table* is deleted after a defined timeout interval. This guarantees that nodes, which are not active anymore or have left the signal range and for this reason their hello message broadcasts are not recieved, get removed. *AGING* can also set a new default gateway e.g. when the current default gateway has left the network or has lost connectivity.

TIMER This is only running in gateway mode. The gateway stores and manages routes to the nodes in the WMN. A local message is sent periodically and caught by the module. There the function *Aging* of the *FBR GW Binary Tree* is called, which deletes obsolete path entries.

ANNOUNCE This is an optional feature. In case of one way traffic as in streaming like e.g. Internet TV the gateway does not receive data package periodically from a node. Thus the recorded route to a node will get outdated and finally deleted. To avoid this, *ANNOUNCE* can be activated so that an announce message is sent periodically to the gateway to refresh the route.

3.3 Kernel Space

Unlike in User Space, the kernel modules differ completely in host and gateway mode. The host module is mainly responsible for traffic inside the WMN i.e. source routing (record route and strict source route). As the gateway is the interface from the WMN to the Internet and vice versa, it has to modify the packages for WMN or Internet usage.

3.3.1 Host module

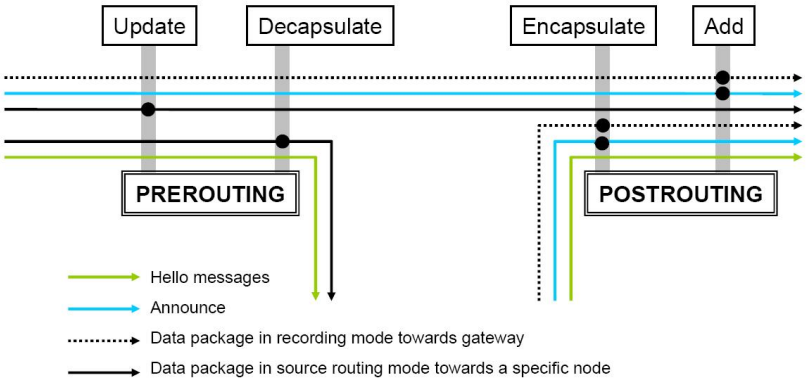


Figure 3: User Mode: Kernel Module

For all data traffic from a mesh node to a gateway a FBR Header is inserted between Network and Transport Layer. This header contains source routing information. When creating a data package this header is included by *Encapsulate*. Nodes on its path to the gateway just modify the FBR Header. They write their IP address into it (Record Route) or update the new destination (strict source routing). For that *Add* and *Update* are called. At its destination the FBR header is removed by *Decapsulate*. All this happens as followed.

The host module uses of two netfilter hooks. *PREROUTING* and *POSTROUTING*. These hooks catch the packages and call functions and finally drop or forward the package. A host in the WMN receives hello messages and data packages. Hello messages pass directly through the module and continue via *Proceed_Received_HELLO*. Data packages for the host are caught in *PREROUTING* and the header is removed by *Decapsulate*. Traffic to a specific node (strict source routing) is also caught and the new destination is set by *Update* and the package is forwarded.

The *POSTROUTING* hook takes care of source routing packages and announces. For that, *Add* is called and the IP address of the host is included in the FBR header. If the host itself creates announce messages or normal traffic *Encapsulate* includes the FBR header. The hello messages which are broadcasted get through the hook without being modified.

3.3.2 Gateway module

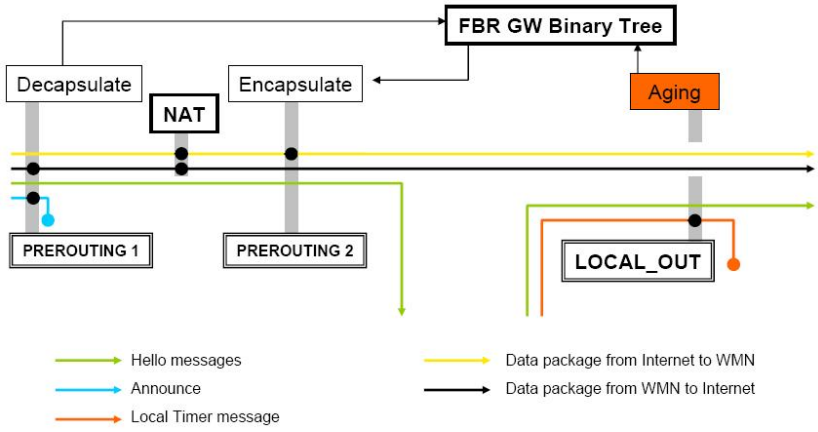


Figure 4: Gateway Mode: Kernel Module

The gateway is the interface between WMN and the Internet. Thus it has to remove the WMN intern FBR header and forward the package to the Internet. The gateway has to prepare also the incoming packages for the WMN. Therefore the specific FBR header containing the route information to a specific host is included. This information is extracted from the *FBR GW Binary Tree*.

In the WMN we assign static IP addresses to the mesh nodes. These ad-

dresses are not valid in the Internet. For this reason the gateway has to use Network Address Translation (NAT) for sending WMN generated packages to the Internet and vice versa. With this procedure a package always contains a valid IP address.

This module consists of three netfilter hooks *PREROUTING 1*, *PREROUTING 2* and *LOCAL_OUT*. Between *PREROUTING 1* and *PREROUTING 2* Network Address Translation (NAT) takes place. To store all the routes to the specific nodes a *FBR GW Binary Tree* is implemented. Sorted by IP addresses (makes searching faster) the FBR headers of all incoming packages are included or refreshed. So the gateway knows all routes to these nodes. To remove obsolete routes an aging mechanism is also implemented.

When an announce message arrives, *PREROUTING 1* drops it and *Decapsulate* refreshes the specific route in *FBR GW Binary Tree*. Data packages to the Internet are forwarded after *PREROUTING 1* has called *Decapsulate*. This has to happen before anything else. Traffic coming from the Internet to the WMN has to pass through NAT first, so its real destination is inserted and is then caught by *PREROUTING 2* and the FBR header is included. Incoming and sent hello messages are not altered.

LOCAL_OUT takes care of the local timer messages created by the user space *Timer*. Every time this package is generated *LOCAL_OUT* drops it and calls *Aging*.

4 Implementation

In this chapter we focus on implementation specific aspects of the FBR protocol. First the environment used is presented, followed by some general remarks about the written code. Then some important data structures are illustrated in detail.

4.1 Environment

The FBR protocol was developed on an IBM T43p Laptop equipped with an Atheros (AR5212) wireless card. The operating system used was a Debian Linux running with Kernel 2.6.14.4. The code was compiled with gcc version 3.3.5 using libc 2.3.2. The implementation was also tested on an IBM T42 Laptop equipped with an Atheros (AR5212) wireless card. The operating system on this computer was a Knoppix v5 with Kernel 2.6.17. The compiling was done with gcc 4.0.4 using libc 2.3.6.

4.2 Code

The written code is based on the AODV-UU implementation specially the timer queue including the corresponding list handling code was reused without major modification. Other functions and structures were modified according to our needs.

The AODV-UU implementation was hardly documented, which made it difficult to get familiar with it. Our implementation, with exception of the timer queue and the list handling, is very well documented. Besides, the functions are written in a way, which is easy to understand, so anybody who has or wants to work with this code should not have a hard time understanding it.

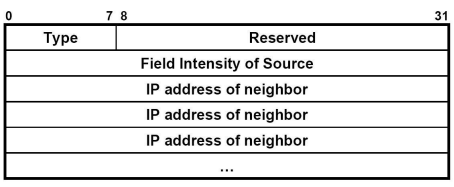
4.3 Structures

Basically there are four important structures in this implementation, the hello packet, the source route header, the routing table and the binary tree which saves the routes to each node in the gateway. First the packet and the header are explained in detail.

4.3.1 Hello Packet

```
#define FBR_MAX_neighbour 15

typedef struct {
    u_int32_t type:8;
    u_int32_t reserved:24;
    u_int32_t field;
    struct in_addr ips[FBR_MAX_neighbour];
} fbr_hello_msg;
```



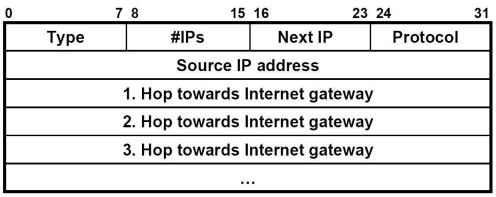
The purpose of the HELLO packet is to inform the surrounding nodes of a certain host about its existence, the current potential it has and a list of those neighbour nodes which have been used to calculate this potential. The transmission of the potential is obviously necessary for its neighbours to calculate

their own field intensity. The list of neighbours is necessary for a process similar to the 'poison reverse' in distance vector routing. If a host sees itself on that list, it will not use this neighbour to calculate the field intensity thus avoiding a swingdown in case of some loss of connectivity. The packet size is dynamic, depending on how many IP addresses the list contains, although a maximum of IP addresses is specified by FBR_MAX_neighbour which is currently set to 15. The HELLO packet structure is also abused for the announce and the timer packets hence the type field.

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 protocol:8;
    u_int32_t ips[FBR_MAX_ROUTE];
};
```



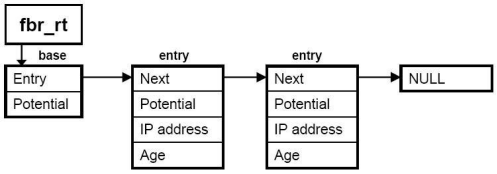
The source routing (SR) struct is actually a header which is inserted into every packet that is transmitted inside the WMN. It is inserted between the IP header [11] and the header on the transport layer, e.g. TCP [12] UDP [13] or ICMP [14]. The SR packet contains all the information the gateway needs to route the packet from the Internet to a specific host in the WMN. It records the nodes it has passed on the way from a host to the gateway. The gateway stores this route and on its way back the SR header contains the whole route. This way every host is able to forward the packet on its way from the gateway to the host (source routing). In detail the packet layout is the same in both, the recording and forwarding mode. It contains the type field which specifies the modes explained before, the quantity of valid IP addresses in the header, a pointer to the slot where either the next recorded IP address can be added or where the IP address of the next hop is stored. Obviously the protocol type of the next higher protocol has to be extracted from the IP header and saved in the SR header before inserting the SR structure into the packet. Last but not least there are FBR_MAX_ROUTE (currently 10) slots available for the route to be saved or inserted. This is a static allocation to avoid the necessity to move around memory at every host the packet passes through.

The HELLO packet and the SR header explained above are compatible, although they exist on different layers. The HELLO message is sending an UDP packet on layer five and the SR header is inserted on layer four. Both packets begin with a type field and all five types mentioned so far are assigned to a different number. This is done in that way so in some future work a motivated person with enough time can combine those two packets on layer four, for example using raw sockets. This would be a much cleaner solution and would solve the unpleasant fact that an announce packet is capsulated inside a SR header, on its way to the gateway.

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

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

fbr_rt_base *fbr_rt;
```



4.3.3 Routing Table

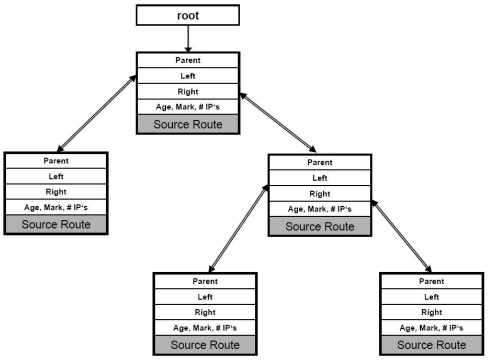
The routing table consists of two parts, the *base* and a linked list of *entries* ordered by their field intensities. Due to the nature of the network the number of neighbours remains small, so a linked list is sufficiently effective. The *base* contains the actual field intensity of the host and it points to the first *entry* of the list of neighbours. Each *entry* contains the IP address of the host which it belongs to and its potential. The potential of every node in the list is put to zero if it used the field intensity broadcasted by this host to calculate its own. The *entries* also contain an aging byte so they can be deleted if they are not updated for some time (missing hello messages). They also contain a pointer to the next *entry* in the list. There is a global pointer to the *base* of the routing table.

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;
    u_int32_t ips[FBR_MAX_ROUTE];
};

struct fbr_bt_node *root;
```



The binary tree is part of the kernel module managed by non-recursive functions called by the netfilter [15] hooks. Its purpose is to save all the routes (ordered by destination) of those hosts which communicate through this gateway. Beside the actual data consisting of the path and the number of hops, each node stores administrative information. Every node has pointers in every direction, an age byte and a visited mark, needed by some algorithms. This data structure is located in the kernel and is very frequently accessed. Besides, the potential amount of information it has to store, is considerable. This raises the need for a more effective data structure than a simple linked list. The effectiveness of this structure in big networks has not been evaluated and stays open. Probably a balanced binary tree or a hash table achieves a higher throughput.

4.4 Definitions

4.4.1 Packet Types

```
#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
```

There are six packet types. Most of them are explained above. The types FBR_TYPE_ANOUNCE and FBR_TYPE_TIMER are used to announce a route to the gateway and to simulate a local timer in kernel space to age the binary tree. This saved us the trouble of implementing a kernel task. One type (FBR_TYPE_LOCAL) is not used. It is a placeholder for somebody who would like to enable local traffic inside the WMN.

4.4.2 Time Intervals

```
#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
```

There is a time interval for each mechanism, sending hellos or announces, or ageing the routing table or the binary tree. The jitter interval defines the time interval in which the hello sending is blurred to avoid repetitive collisions.

4.5 Hooks

There are two modules in our implementation one for a gateway and one for a host. Because their functionality differs, each uses a different set of netfilter hooks. The host module has a prerouting and a postrouting hook with high and normal priority respectively. The gateway module has three hooks, two prerouting and one local out hook. The two prerouting hooks have high and normal priority, the reason therefore is the NAT [16] table which is in between those priorities.

4.6 Files

main{.c} starts the daemon and loads the modules

defs{.h} header file with some important definitions

fbr_socket{.c/.h} manages the socket to send udp packets

fbr_hello{.c/.h} constructs, sends and receives hello packets

fbr_announce{.c/.h} sends announces to the gw

fbr_bt_timer{.c/.h} sends timer messages to the netfilter hook

list{.c/.h} some list manipulation functions

timer_queue{.c/.h} defines the timer and handles them

nl{.c/.h} manages the netlink socket to communicate with the kernel routing table

routing_table{.c/.h} constructs and manages the internal routing table

mod_gw/fbr-mod-gw{.c} loads the module, registering some netfilter hooks

mod_host/fbr-mod{.h} some definitions for the module in general

mod_gw/fbr-cap{.c/.h} manages the source routing layer

mod_gw/fbr-bt{.c/.h} manages the binary tree

mod_host/fbr-mod-host{.c} loads the module, registering some netfilter hooks

mod_host/fbr-mod{.h} some definitions for the module in general

mod_host/fbr-cap{.c/.h} manages the source routing layer

5 Evaluation

To evaluate the FBR Protocol, we have tested four different scenarios. We have checked the following functionalities: source routing, dynamic neighbour choosing and gateway switching. The evaluation took place in the ETL/ETZ building at ETH Zurich.

As we wanted to compare the different evaluations we pinged 209.85.135.104 (belongs to Google) in every experiment. As a reference we first measured the round trip time (RTT) directly from the gateway. The average ping RTT is 14.8, see PING RESULTS FROM GW.

```
# PING RESULTS FROM GW
PING 209.85.135.104 (209.85.135.104) 56(84) bytes of data.
64 bytes from 209.85.135.104: icmp_seq=1 ttl=243 time=14.8 ms
64 bytes from 209.85.135.104: icmp_seq=2 ttl=243 time=14.7 ms
64 bytes from 209.85.135.104: icmp_seq=3 ttl=243 time=14.7 ms
...
64 bytes from 209.85.135.104: icmp_seq=93 ttl=243 time=14.7 ms
64 bytes from 209.85.135.104: icmp_seq=94 ttl=243 time=14.7 ms
64 bytes from 209.85.135.104: icmp_seq=95 ttl=243 time=14.8 ms

--- 209.85.135.104 ping statistics ---
95 packets transmitted, 95 received, 0% packet loss, time 94707ms
rtt min/avg/max/mdev = 14.576/14.761/15.088/0.133 ms
```

5.1 Source Routing

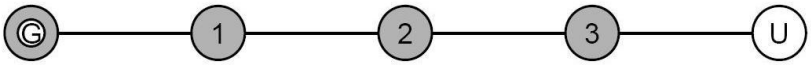


Figure 5: Chain of five nodes from Internet gateway (left) to User U (right)

To verify the source routing function i.e. record route and strict source routing, a chain of five nodes is tested as shown in Figure 5. The User U pings and the data packages traverse along node 3, 2, 1 and Internet gateway and vice versa. The results are captured in PING RESULTS SOURCE ROUTING. The average ping RTT is 56.6 ms and we measure a packet loss of 5%.

```
# PING RESULTS SOURCE ROUTING
PING 209.85.135.104 (209.85.135.104) 56(84) bytes of data.
64 bytes from 209.85.135.104: icmp_seq=2 ttl=239 time=78.6 ms
64 bytes from 209.85.135.104: icmp_seq=3 ttl=239 time=35.7 ms
64 bytes from 209.85.135.104: icmp_seq=4 ttl=239 time=30.0 ms
...
64 bytes from 209.85.135.104: icmp_seq=107 ttl=239 time=28.3 ms
64 bytes from 209.85.135.104: icmp_seq=108 ttl=239 time=33.3 ms
64 bytes from 209.85.135.104: icmp_seq=109 ttl=239 time=55.3 ms

--- 209.85.135.104 ping statistics ---
110 packets transmitted, 104 received, 5% packet loss, time 109737ms
rtt min/avg/max/mdev = 23.296/56.618/234.093/35.301 ms
```

In addition we also ping from node 3, 2 and 1 in Figure 5 to receive some metric information. The result is shown below.

```
# PING from node 3
--- 209.85.135.104 ping statistics ---
184 packets transmitted, 176 received, 4% packet loss, time 184173ms
rtt min/avg/max/mdev = 19.356/48.156/209.941/34.605 ms
```

```
# PING from node 2
--- 209.85.135.104 ping statistics ---
138 packets transmitted, 130 received, 5% packet loss, time 137839ms
rtt min/avg/max/mdev = 19.722/45.763/159.706/27.790 ms

# PING from node 1
--- 209.85.135.104 ping statistics ---
72 packets transmitted, 69 received, 4% packet loss, time 71034ms
rtt min/avg/max/mdev = 15.307/26.960/130.472/23.973 ms
```

Remarkable is that the packet loss occurs mainly between node 1 and the Internet gateway, because they are working at the limit where they still have a signal reception of each other. Inside the WMN no significant packet loss happens during stable state. When the nodes are moving as they do in the following evaluation tests, other results can be observed with increased packet loss.

5.2 Dynamic neighbour Selection

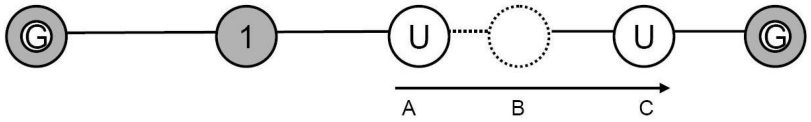


Figure 6: WMN with two Internet gateways for dynamic neighbour selection

To test if FBR also works with two Internet gateways, we arrange a WMN as shown in Figure 6. We start at position A and move towards position C, the default gateway changing takes place at position B. There, node 1 and the right Internet gateway are in reach and so, due to the higher potential of the Internet gateway the change happens fast. This means that the change is based on incoming hellos from another node with higher potential and the new default gateway is set immediately.

The average RTT is 27.9 ms and we have no packet loss, see PING RESULTS DYNAMIC NEIGHBOUR SELECTION. At position B our default gateway is the right Internet gateway. All ping requests which has been sent via node 1 and the left Internet gateway also arrive at us because we are still in reach of node 1 when arriving at position C.

```
# PING RESULTS DYNAMIC NEIGHBOUR SELECTION
PING 209.85.135.104 (209.85.135.104) 56(84) bytes of data.
64 bytes from 209.85.135.104: icmp_seq=1 ttl=241 time=20.7 ms
64 bytes from 209.85.135.104: icmp_seq=2 ttl=241 time=25.4 ms
64 bytes from 209.85.135.104: icmp_seq=3 ttl=241 time=40.4 ms
...
64 bytes from 209.85.135.104: icmp_seq=15 ttl=241 time=33.0 ms
64 bytes from 209.85.135.104: icmp_seq=16 ttl=241 time=36.5 ms
64 bytes from 209.85.135.104: icmp_seq=17 ttl=242 time=16.8 ms
64 bytes from 209.85.135.104: icmp_seq=18 ttl=242 time=16.7 ms
64 bytes from 209.85.135.104: icmp_seq=19 ttl=242 time=17.7 ms
...
64 bytes from 209.85.135.104: icmp_seq=45 ttl=242 time=18.1 ms
64 bytes from 209.85.135.104: icmp_seq=46 ttl=242 time=30.8 ms
64 bytes from 209.85.135.104: icmp_seq=47 ttl=242 time=46.1 ms

--- 209.85.135.104 ping statistics ---
47 packets transmitted, 47 received, 0% packet loss, time 46315ms
rtt min/avg/max/mdev = 16.628/27.953/130.032/18.902 ms
```

5.3 Shut down of an Internet gateway

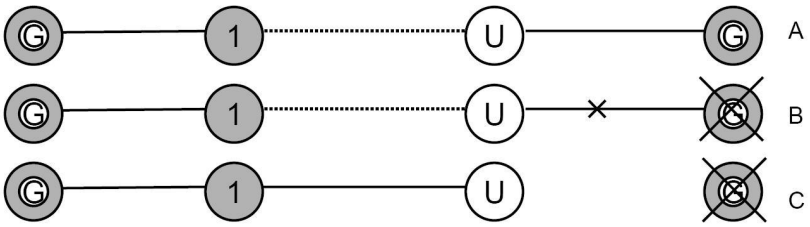


Figure 7: WMN with two Internet gateways

A similar test as the last, but here we test another functionality. See Figure 7. Node U has node 1 and the right Internet gateway as neighbours and so all traffic goes over the right Internet gateway. We shut down the right Internet gateway (B) and via the aging mechanism node 1 choses (C) as the new default gateway. On the contrary to the previous experiment, this change happens slowly. First the hello timeout has to pass and afterwards the new default gateway is set. This explains the extra package loss of packet 31 to 41, see PING RESULTS SHUT DOWN GW. As a future work we suggest to consider the MAC feedback to see if the package has been sent.

```
# PING RESULTS SHUT DOWN GW
PING 209.85.135.104 (209.85.135.104) 56(84) bytes of data.
64 bytes from 209.85.135.104: icmp_seq=1 ttl=242 time=36.2 ms
64 bytes from 209.85.135.104: icmp_seq=2 ttl=242 time=51.6 ms
64 bytes from 209.85.135.104: icmp_seq=3 ttl=242 time=16.8 ms
...
64 bytes from 209.85.135.104: icmp_seq=28 ttl=242 time=19.6 ms
64 bytes from 209.85.135.104: icmp_seq=29 ttl=242 time=35.0 ms
64 bytes from 209.85.135.104: icmp_seq=30 ttl=242 time=17.1 ms //shut down
64 bytes from 209.85.135.104: icmp_seq=42 ttl=241 time=1051 ms
64 bytes from 209.85.135.104: icmp_seq=43 ttl=241 time=124 ms
...
64 bytes from 209.85.135.104: icmp_seq=76 ttl=241 time=32.2 ms
64 bytes from 209.85.135.104: icmp_seq=77 ttl=241 time=54.7 ms
64 bytes from 209.85.135.104: icmp_seq=78 ttl=241 time=67.0 ms

--- 209.85.135.104 ping statistics ---
80 packets transmitted, 67 received, 16% packet loss, time 79357ms
rtt min/avg/max/mdev = 16.592/112.499/1051.716/187.938 ms, pipe 2
```

5.4 Walk through WMN

To test if the correct neighbour is selected as default gateway in every scenario, we arranged a WMN as in Figure 8. Node 1 has the Internet gateway and nodes 2 and 3 as neighbours. The Nodes 2 and 3 do not see each other. Our walk begins at position A where we (Node U) have connectivity over node 2. There we start to ping 209.85.135.104. Moving from A towards B we also get in range of node 3, but our default gateway is still Node 2 because the potential of node 2 and 3 are equal and no gateway changing is necessary.

Reaching position C where the connection to Node 2 fails, we change our default gateway to Node 3. That happens at about icmp_seq=45. The changing of the default gateway carries along some extra packet loss as shown in PING RESULTS WMN WALK. Detailed procedure: Three to four seconds after losing connectivity to node 2, the aging mechanism intervenes due to missing hello messages from node 2, removes it and sets node 3 as default gateway. All packages in this time interval are lost.

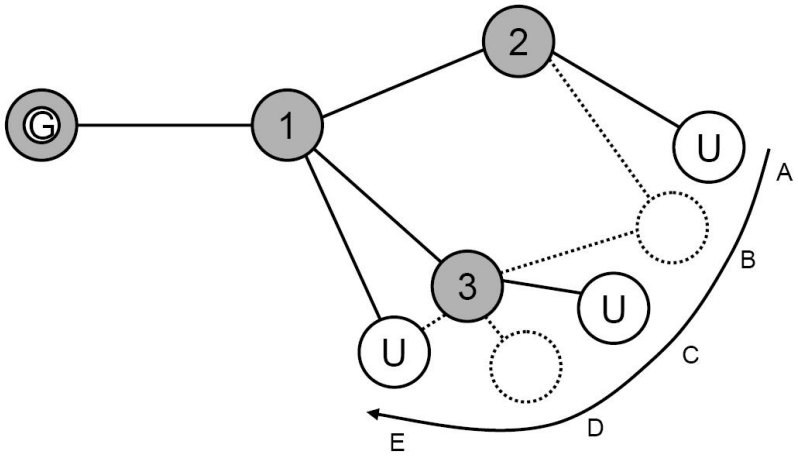


Figure 8: WMN with Internet gateway and four nodes

Passing position D we reach position E. There we get connectivity to node 3 and 1. Obviously the default gateway is changed to Node 1 as its emitting potential is higher than the one of Node 3. Opposite to the other default gateway change, this time the change happens fast as it effects due to incoming hellos from a higher potential node and not via the aging mechanism. That happens at about icmp_seq=71 and again some extra packet loss occurres. The packet loss and the long RTT as in icmp_seq=76 is based on the weak signal when node 1 is chosen as default gateway. A hello message is sent in broadcast mode thus it has a larger reach as an unicast package. To avoid such packet loss we suggest as future work to consider the signal ratio when choosing a new default gateway. The average RTT is 77.4 ms and we have a total packet loss 10%.

```
# PING RESULTS WMN
PING 209.85.135.104 (209.85.135.104) 56(84) bytes of data.
64 bytes from 209.85.135.104: icmp_seq=1 ttl=240 time=81.9 ms
64 bytes from 209.85.135.104: icmp_seq=2 ttl=240 time=81.0 ms
64 bytes from 209.85.135.104: icmp_seq=3 ttl=240 time=20.3 ms
...
64 bytes from 209.85.135.104: icmp_seq=38 ttl=240 time=87.4 ms
64 bytes from 209.85.135.104: icmp_seq=39 ttl=240 time=18.3 ms//slow change
64 bytes from 209.85.135.104: icmp_seq=43 ttl=240 time=32.8 ms//node 2 -> node 3
64 bytes from 209.85.135.104: icmp_seq=44 ttl=240 time=21.2 ms
64 bytes from 209.85.135.104: icmp_seq=45 ttl=240 time=19.1 ms
...
64 bytes from 209.85.135.104: icmp_seq=68 ttl=240 time=21.7 ms
64 bytes from 209.85.135.104: icmp_seq=69 ttl=240 time=28.5 ms
64 bytes from 209.85.135.104: icmp_seq=70 ttl=240 time=17.1 ms//fast change
64 bytes from 209.85.135.104: icmp_seq=71 ttl=240 time=16.4 ms//node 3 -> node 1
64 bytes from 209.85.135.104: icmp_seq=76 ttl=241 time=2071 ms
64 bytes from 209.85.135.104: icmp_seq=77 ttl=241 time=1088 ms
64 bytes from 209.85.135.104: icmp_seq=79 ttl=241 time=35.4 ms
64 bytes from 209.85.135.104: icmp_seq=80 ttl=241 time=35.8 ms
...
64 bytes from 209.85.135.104: icmp_seq=94 ttl=241 time=19.0 ms
64 bytes from 209.85.135.104: icmp_seq=95 ttl=241 time=18.7 ms
64 bytes from 209.85.135.104: icmp_seq=96 ttl=241 time=18.9 ms

--- 209.85.135.104 ping statistics ---
96 packets transmitted, 86 received, 10% packet loss, time 95512ms
rtt min/avg/max/mdev = 16.065/77.434/2071.040/245.700 ms, pipe 3
```

6 Conclusion

The Communication Systems Group of the Computer Engineering and Networks Laboratory at ETH Zurich developed an approach called Field Based Routing for a scalable routing protocol for Wireless Mesh Networks. Our contribution was to implement such a protocol on Linux 2.6.

Therefore we evaluated several existing ad-hoc routing protocols to get information on how they are generally implemented. We then decided to build up FBR onto AODV-UU, which is a reactive routing protocol for ad-hoc networks. AODV-UU contains some mechanism or approaches which we could modify and reuse.

We designed FBR as a user space daemon and two kernel modules (host and gateway mode). The host module takes care of intern WMN traffic and supports source routing e.g. record route and strict source route. As the gateway is the interface to the Internet it prepares the data packages from the WMN to enter the Internet and vice versa.

Our Evaluation has showed that the implementation works successfully. We tested FBR in several scenarios and it did behave as requested.

FBR will be available as open source software.

7 Future Work

As future work we will count up things we think might improve FBR or make it more scalable and powerful.

Auto-configuration of IP addresses For using FBR we had to set a static IP address, netmask and cell to each device. An implementation of a service like DHCP or Link Local IPv6 address in Wireless Mesh Network would solve this and FBR would be easier to operate.

Selection of preferred neighbours based on signal ratio FBR just chooses the neighbour with the highest potential as default gateway no matter of physical position of this neighbour. To guarantee that no device just a few meters to a node is chosen as default gateway nor the one who is hardly in reach, the signal ratio of the hello messages could be taken into account to choose the optimal neighbour as default gateway.

Adaption of gateway potential based on load Dependent of a gateways state (lot of traffic, idle) it could increase or decrease its emitting potential to reach some distribution of the network data traffic.

IPV6 Enable IPV6 for FBR.

Large Scale Evaluations Do an evaluation with 100 or 1000 participants.

Dynamic length of source routing header Make the source routing header dynamic, now it is set to a static size.

MAC layer feedback Consider the MAC layer feedback to see if packages are really sent.

Data structure The hello messages are sent via UDP. To equate the hello messages to the other WMN traffic, one could implement hello messages to work also with the SR header

8 Bibliography

References

[1] Ian F. Akyildiz and Xudong Wang and Weilin Wang. Wireless Mesh Networks: A Survey. *Computer Networks Journal (Elsevier)*, 2005.

[2] Thomas Staub. *Ad-hoc and Hybrid Networks: Performance Comparison of MANET Routing Protocols in Ad-hoc and Hybrid Networks*. Institute of Computer Science and Applied Mathematics, University of Berne, 2004.

[3] D. Waitzman, C. Partridge, and S.E. Deering. Distance Vector Multicast Routing Protocol. RFC 1075 (Experimental), November 1988.

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

[5] 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.

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

[7] Sung-Ju Lee, Elizabeth M. Belding-Rover, and Charles E. Perkins. *Ad-hoc On-Demand Distance-Vector Routing Scalability*. Internet Systems Storage Lab, Hewlett-Packard Laboratories, Palo Alto, CA, USA, 2002.

[8] Erik Nordstroem. Ad hoc On-Demand Distance Vector Routing. http://core.it.uu.se/core/index.php/AODV-UU, December 2006.

[9] Luke Klein-Berndt. *kernel-aodv*. National Institute of Standards and Technology (NIST), 2005.

[10] Erik Nordstroem. DSR UU. http://core.it.uu.se/core/index.php/DSR-UU, December 2006.

[11] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[12] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[13] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[14] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFC 950.

[15] NETFILTER. www.netfilter.org.

[16] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

A Howto FBR

A.1 How to install the fbr 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
```

Now everything is ready to compile the fbr protocol implementation:

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

Than start the device:

```
# 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.

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.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
```

B CD Content

On the CD you find the following folders

Pingfiles Contains all ping files we have mentioned in this thesis.

Presentation Contains the presentation we did at December 21. 2006

Thesis Contains .tex file, pictures and a pdf version of this thesis.

Software Contains our FBR implementation and the software which is mentioned in Related Work.