

---

Semester Thesis August 3, 2006

---

# Implementing a virtual network interface for Linux 2.6

**Silvan Graf**  
(D-INFK)

---

Advisor: **Patrick Stuedi**

Prof. G. Alonso  
Institute of Pervasive Computing  
ETH Zürich



# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	Transparent heterogeneous mobile ad hoc networks . . . . .	3
<b>3</b>	<b>Related Work</b>	<b>5</b>
3.1	Transparent heterogeneous mobile ad hoc networks . . . . .	5
3.2	Linux kernel networking components . . . . .	5
3.2.1	Linux Ethernet Bridge . . . . .	5
3.2.2	Netfilter . . . . .	5
3.3	Routing protocols for MANET . . . . .	5
3.3.1	AODV . . . . .	6
3.3.2	OLSR . . . . .	6
<b>4</b>	<b>An introduction to Linux 2.6 kernel programming</b>	<b>7</b>
4.1	The Linux kernel . . . . .	7
4.2	The Hello World Module . . . . .	7
4.3	The build system . . . . .	9
4.3.1	Becoming part of the kernel . . . . .	9
4.4	The Linux device model . . . . .	11
4.4.1	Registering with the device model . . . . .	11
4.4.2	System FS . . . . .	13
4.5	The Network subsystem . . . . .	16
4.5.1	Initialization . . . . .	16
4.5.2	Default interface . . . . .	17
4.5.3	Packet transport . . . . .	18
<b>5</b>	<b>Implementation of the virtual interface</b>	<b>21</b>
5.1	The kernel module . . . . .	21
5.1.1	User interface . . . . .	21
5.1.2	Registering with the device model . . . . .	22
5.1.3	The hook in the networking stack . . . . .	22
5.1.4	The neighbor database . . . . .	24
5.1.5	Processing incoming packets . . . . .	25
5.1.6	Processing outgoing packets . . . . .	25
5.2	The libvi library . . . . .	26
5.3	The victl command . . . . .	26
<b>6</b>	<b>Experiments and Results</b>	<b>27</b>
6.1	Throughput . . . . .	27
6.2	Handover . . . . .	29

<b>7</b>	<b>Users' guide</b>	<b>31</b>
7.1	Installation . . . . .	31
7.2	Usage . . . . .	32
<b>8</b>	<b>Conclusions and Future Work</b>	<b>33</b>
	<b>Bibliography</b>	<b>34</b>

# Chapter 1

## Abstract

This semester thesis shows how a virtual network interface can be used to transparently build heterogeneous wireless multihop networks with varying MAC level protocols. The implementation of such an interface for Linux 2.6 and the usage in cooperation with AODV and OLSR are shown. Moreover, performance measurements are presented indicating the overhead introduced by the virtual interface with respect to throughput and handover time. The thesis further contains an introduction into Linux kernel module programming.



# Chapter 2

## Introduction

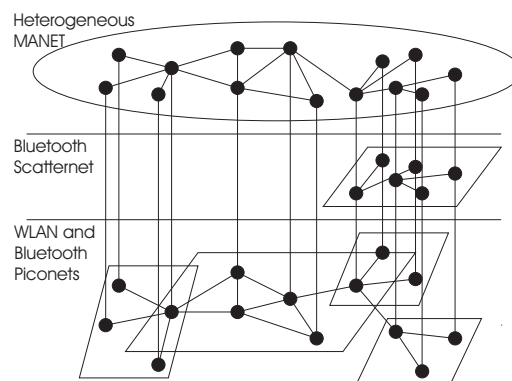
This semester thesis has been motivated by a previous work on transparent heterogeneous mobile ad hoc networks [8]. In this work it has been shown how a virtual network interface providing a MAC layer similar to 802.11 and a broadcast emulation can be used to transparently build wireless multihop networks with varying MAC level protocols. While [8] is based on a modified Linux Ethernet Bridge on kernel 2.4, this work started from scratch on kernel 2.6.

This thesis presents how a virtual network interface providing a flexible handover infrastructure, end-to-end communication abstraction and a broadcast emulation layer is implemented as a kernel module for Linux 2.6. A chapter on Linux kernel module programming introduces the reader to the key concepts encountered in the implementation of the virtual network interface. A userspace library and a tool to configure the virtual interface are provided. A guide on how to use the virtual interface together with the routing protocols AODV and OLSR is given. Various experiments and their results are discussed to show the efficiency of this approach.

### 2.1 Transparent heterogeneous mobile ad hoc networks

Mobile ad hoc networks already exist in many forms of which the most popular incarnations are personal area networks (PAN) and wireless sensor networks. For many applications it is desirable to transparently build a heterogeneous mobile ad hoc network consisting of different MANETs.

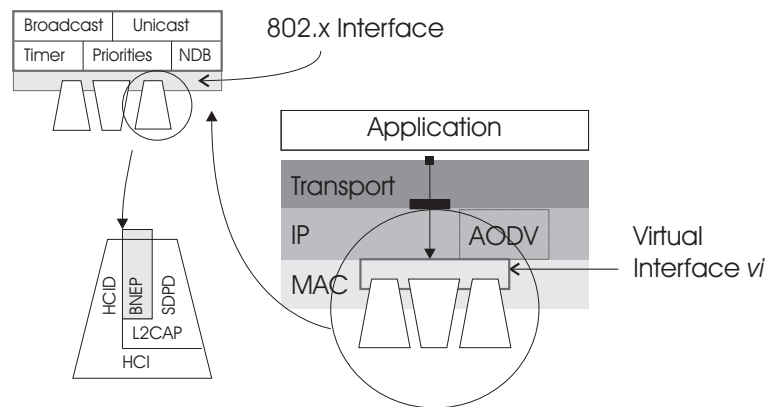
**Figure 2.1:** Heterogeneous Mobile Ad Hoc Network



In figure 2.1 it is visible how nodes participating in different MANETs join those together to form a single transparent mobile ad hoc network.

The different usage scenarios impose their very specific requirements on devices and protocols. Bluetooth PAN and WSN require the devices to be energy efficient whereas a 802.11 wireless network in an office needs a high throughput. The protocol stacks for the different network types are optimized to their use and the protocols therefore differ strongly. Most research on MANET assumes these networks to have a common MAC schema. The virtual interface approach combines devices with different MAC level protocols into one interface. Incoming packets on the physical devices are intercepted and further processed by the virtual interface. The virtual interface sends outgoing packets over the appropriate physical interface (cf. [8], 5.1.5 or 5.1.6). Figure 2.2 shows how a Bluetooth BNEP connection is embedded into the virtual interface. A packet passes up the Bluetooth stack and is intercepted by the virtual interface just before entering the IP layer.

**Figure 2.2:** Virtual Interface



# Chapter 3

## Related Work

### 3.1 Transparent heterogeneous mobile ad hoc networks

In this work[8] a modified Linux Ethernet Bridge on kernel 2.4 was used to demonstrate how wireless multihop networks can be built using a virtual interface providing a MAX layer similar to 802.11 and broadcast emulation<sup>1</sup> The ideas of that work were used to implement the virtual interface for Linux 2.6.

### 3.2 Linux kernel networking components

#### 3.2.1 Linux Ethernet Bridge

The Linux Ethernet Bridge allows to put several real interfaces into a virtual bridging device. It is not only an in-kernel equivalent to a real ethernet bridge but together with Netfilter a very sophisticated tool for packet filtering.

#### 3.2.2 Netfilter

Netfilter and its userspace administration tool iptables are the packet filter firewalling toolkit for Linux. The Linux kernel contains many so-called Netfilter targets to build powerful packet filter rulesets. Netfilter can intercept packets at many states of their processing and perform arbitrary operations on them.

The Netfilter target `ip_queue` passes packets from kernel to userspace and back which opens up many new applications.

### 3.3 Routing protocols for MANET

The routing in mobile ad hoc networks necessitates specialized algorithms and protocols that can cope with the dynamic nature of appearing and vanishing neighbours. Two major protocols have been used in this work to test the virtual interface in a realistic environment.

---

<sup>1</sup>The Bluetooth Network Encapsulation Protocol BNEP provides a point to point connection. A broadcast is to be understood as sending a packet over all existing BNEP links.

### 3.3.1 AODV

The Ad Hoc On demand Distance Vector routing protocol is a reactive routing algorithm for MANET, meaning that routes between nodes are only built as soon as and maintained as long as needed by a source node. AODV works with both unicast and multicast networks, whereas in the latter it builds trees of multicast group members and nodes that connect the multicast members. An implementation of AODV for Linux systems named AODV-UU is provided by the Uppsala University[1]. It consists of a kernel module and a userspace daemon. Although still experimental, aodvd performs well in a Linux MANET.

AODV-UU uses Netfilter to capture the packets. It implements a facility similar to `ip_queue` to pass the packets to userspace. The userspace daemon then processes the packets and modifies the kernel routing table.

### 3.3.2 OLSR

The Optimized Link State Routing protocol is a proactive, table-driven routing algorithm for MANET. An implementation of the OLSR protocol is provided by[6]. Olsrd runs as a standalone server process and is platform independent. It is supposed to work on Linux, FreeBSD, NetBSD, OS X and even on Windows. Olsrd does not depend on a kernel module. All operations are performed in userspace. Explicit interaction with the kernel is only necessary to manipulate the routing table.

## Chapter 4

# An introduction to Linux 2.6 kernel programming

This chapter tries to introduce the experienced systems programmer with a profound knowledge in C to a few of the basic concepts of Linux kernel programming. The reader is assumed to be familiar with using the GNU toolchain, namely *gcc* and *make*, further should he know how to build a customized kernel.

It is of course not possible to give an elaborate introduction to kernel programming in such a thesis. This introduction is supposed to teach the reader the very basics of Linux kernel programming. It shows the major aspects related to the implementation of the virtual network interface, namely to get a module built and have it registered with the central facilities of the Linux kernel.

A more in-depth look at Linux kernel programming is given in [3] or [4].

Any structure or function which is referenced in the following can be looked up at the "Cross-Referencing Linux"[5] project. They provide a search engine and a hyperlinked view of the Linux source code.

### 4.1 The Linux kernel

The Linux kernel is a so-called monolithic kernel, i.e. all operating system services such as memory and process management, hardware drivers, networking and concurrency are implemented as a whole and run in supervisor mode sharing the same address space. Linux provides the ability to load so-called modules at run-time. These become part of the kernel as if they were linked-in.

Most device drivers are implemented as modules, although many of them can be linked into the kernel at compile-time. The decision whether to link a driver into the kernel or to have it as a module is based on the actual needs. The modern way is to have all drivers which are not needed at an early phase of the boot process loaded as modules when needed.

### 4.2 The Hello World Module

Following the tradition of most programming related texts the first example will print "Hello, world". This example uses the logging facility of the kernel. The output is visible either on the console, in the *dmesg* output or in the *syslog*, i.e. usually this is */var/log/messages*.

Listing 4.1 shows the implementation of this simple module. Even in this simplistic example a peculiarity of the Linux kernel shows up: the heavy usage of preprocessor macros. The first is found after the includes. *MODULE\_LICENSE* declares the license under which

a module is distributed. As the Linux kernel itself is distributed only under GPL [2] which does not allow linking proprietary objects to GPL-objects, it is quite unavoidable to choose GPL as the module's license. Otherwise many kernel features are hidden from the module, which is very restricting.

The next macro is encountered in line 7. *KERN\_ALERT*, its friends *KERN\_DEBUG*, *KERN\_INFO* and others define the class of a log message which is to be printed to the kernel log ring-buffer. These macros expand to strings which prefix the actual message.

The last two lines of this example tell the kernel how to load and unload this module, again these are macros.

Listing 4.2 shows the corresponding makefile and listing 4.3 how the module is loaded into the kernel. Looking at the output of the *dmmsg* program should reveal the two strings.

**Listing 4.1:** A minimal kernel module

```

1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("GPL");
4
5 static int hello_init(void)
6 {
7     printk(KERN_ALERT "Hello , world\n");
8     return 0;
9 }
10
11 static int hello_exit(void)
12 {
13     printk(KERN_ALERT "Goodbye , world\n");
14 }
15
16 module_init(hello_init);
17 module_exit(hello_exit);

```

**Listing 4.2:** The Makefile

```

1 ifneq (($KERNELRELEASE),)
2     obj-m := hello.o
3 else
4     KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5     PWD := $(shell pwd)
6
7 default:
8     $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
9 endif

```

**Listing 4.3:** Building and loading the module

```

1 make
2 insmod hello.ko
3 rmmod hello

```

## 4.3 The build system

The previous section showed how an external module can be built. This build process already involves the kernel build system. If a module is to be distributed as part of the kernel, its interaction goes further. The kernel build system not only comprehends compiling and linking but also the configuration. The user usually configures the kernel options through *make config* or its derivatives *make menuconfig* or *make xconfig*. These tools allow the user to select which modules should make part of the kernel and how they will be linked to it - statically or as a module. A number of other parameters can be adjusted during this process. To make a module appear in these tools, it obviously needs to announce its presence.

### 4.3.1 Becoming part of the kernel

To let the module appear in the network section of the kernel configuration, its source has to be moved to *net/hello/* in the kernel source tree. The file *net/Kconfig* has to contain a line *source "net/hello/Kconfig"*. In the *hello* directory, a new file *Kconfig* has to be created according to listing 4.4. The makefile has to be adapted to its new environment as the

**Listing 4.4:** Config file for the kernel build system

```
1 config HELLO
2     tristate "Hello world module"
3     ---help---
4         To compile this code as a module ,
5         choose M here: the module
6         will be called hello .
7
8         If unsure , say N.
```

module has to be compiled if and only if it is enabled in the configuration. Listing 4.5 shows how it might look like. Apart from some general conventions for in-kernel makefiles, the main difference to the simple one in the previous section is in line 15 where the content of the variable *CONFIG\_HELLO* is evaluated. This variable is set by the configuration system of the kernel and refers to the *config HELLO* directive in listing 4.4.

Listing 4.5: Makefile using the kernel build system

```

1 DEBUG = y
2
3 ifeq ($(DEBUG),y)
4     DEBFLAGS = -O -g -DHELLO_DEBUG
5 else
6     DEBFLAGS = -O2
7 endif
8
9 CFLAGS += $(DEBFLAGS) -I$(LDDINC)
10
11 TARGET = hello
12
13 ifneq ($(KERNELRELEASE),)
14
15 obj-$(CONFIG_HELLO)      := hello.o
16
17 #hello-objs := #no other objects are linked to hello.o
18
19 else
20
21 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
22 PWD       := $(shell pwd)
23
24 modules:
25     $(MAKE) -C $(KERNELDIR) M=$(PWD) LDDINC=$(PWD) modules
26
27 endif
28
29
30 install:
31     install -d $(INSTALLDIR)
32     install -c $(TARGET).o $(INSTALLDIR)
33
34 clean:
35     rm -rf *.o *~ core .depend *.ko
36     rm -rf *.mod.c .tmp_versions *.cmd
37
38
39 depend .depend dep:
40     $(CC) $(CFLAGS) -M *.c > .depend
41
42 ifeq (.depend,$(wildcard .depend))
43 include .depend
44 endif

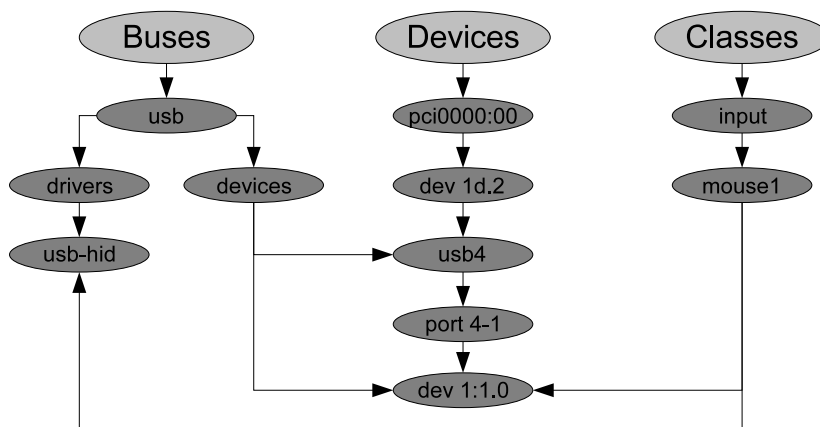
```

## 4.4 The Linux device model

Linux 2.6 introduces a unified device model, a single data structure containing all the information on how the system is put together. Advanced features like hotplugging devices on USB and PCI or power management demanded for a more sophisticated design than the one in Linux 2.4.

The kernel programmer's work became easier because all subsystems work similarly while at the same time the registration of a driver and its devices to the system may have become more difficult. Certainly, the new device model is a giant step in the development of the Linux kernel and shows that Linux 2.6 is a modern and well designed operating system. The device model is mainly split into buses, classes and devices. A small piece of

Figure 4.1: A look into the device model



it is shown in figure 4.1 which is adapted from [3].

A bus represents the way a device is connected to the system, whereas classes group devices according to their function. Two network devices, connected to the PCI bus and the USB bus, respectively, appear in the same class as they provide the same function.

Each object in the device model - e.g. device, driver, bus - is represented by a *kobject*. The *kobject*'s tasks include reference counting, SysFS representation, hotplug event handling. It holds the device module structure together by having pointers to the parent, a *kset*, a list containing its children. It helps distinguishing the different types of *kobjects* with a pointer to a *kobj\_type*.

The *kobject* structure and its primary helpers *kobj\_type* and *kset* are defined in *include/linux/kobject.h*. An excerpt of this file is given in listing 4.6.

Listing 4.7 shows how a *kobject* is normally used. It is a member of the struct which wants to use *kobject*'s facilities. This technique is encountered throughout the kernel in many places, e.g. the linked list implementation (cf. listing 4.6). At this point it might be important to know that the basic structure *device* of which any device in the kernel has an instance normally is not used solely, instead each subsystem defines a container for this structure. Another interesting point is to see that the structure *class\_device* contains a *kobject* and the structure *device* contains another one. The usefulness of this will be seen in the section covering SystemFS.

### 4.4.1 Registering with the device model

Devices in Linux 2.6 normally won't be created out of the blue. The need for a *device* structure or even the whole driver arises when a device is detected through hotplug events

Listing 4.6: *kobject* structures

```
1 struct kobj_type {
2     void (*release)(struct kobject *);
3     struct sysfs_ops * sysfs_ops;
4     struct attribute ** default_attrs;
5 };
6 struct kset {
7     struct subsystem * subsys;
8     struct kobj_type * ktype;
9     struct list_head list;
10    struct kobject kobj;
11    struct kset_hotplug_ops * hotplug_ops;
12 };
13 struct kobject {
14     char * k_name;
15     char name[KOBJ_NAME_LEN];
16     struct kref kref;
17     struct list_head entry;
18     struct kobject * parent;
19     struct kset * kset;
20     struct kobj_type * ktype;
21     struct dentry * dentry;
22 };
```

Listing 4.7: A *kobject* consumer

```
1 struct class_device {
2     struct list_head node;
3     struct kobject kobj;
4     struct class * class;
5     struct device * dev;
6     void * class_data;
7     char class_id[BUS_ID_SIZE];
8 };
```

or probing on the bus. Only special devices like the pure virtual network interface have to be initialized and registered with their respective subsystems manually. Another exception are busses which define on the one hand a *bus\_type* and on the other hand a *device*. Busses like the platform bus which do not have a physical representation (e.g. the USB has a representation in form of a UHCI controller) have to initialize their *device* structures themselves.

Network devices register with the network system through *register\_netdevice* which also includes a registration with the class *net*. It is important to distinguish between the different institutions to which a device might register and to know the various structures needed to do this. Table 4.1 tries to give an overview of the major entry points to the device model. It shows the relationship of subsystems and their structures and registration functions.

**Table 4.1:** Registration facilities of the device model

part of the model	structure	function
class	struct <i>class_device</i>	<i>class_device_register</i>
network subsystem	struct <i>net_device</i>	<i>register_netdevice</i>
bus	struct <i>device</i>	<i>device_register</i>

#### 4.4.2 System FS

The user interface to the new device model is a filesystem which is usually mounted in */sys*. It lists all devices, drivers, busses and their relations. All of them can export attributes which then are listed as files. Links to other parts of the system are implemented as directories, i.e. the pci bus contains directories representing the actual busses (pci controller) which again contain links to the connected devices. Device directories contain links to their drivers. Via this filesystem the user or system utilities can access and modify the parameters of devices and drivers. The filesystem behaves much like the old */proc* filesystem, although its structure is quite strictly defined. The user can manipulate it using *echo*, *cat* and similar tools. The library *libvi* and the management tool *victl* make use of the files in the *sys* filesystem. Looking at listing 4.8 one might be able to detect the correlation of the SysFS tree and the data structure showed in figure 4.1. As attributes are exported to SysFS as files, one has to define functions for *read* and *write* operations. The kernel provides macros and a convenient API to decorate a *kobject* with custom attributes. A directory containing several attributes is attached to an existing *kobject* as shown in listing 4.9. The function *add\_myattrs* is usually called upon initialization of the *class\_device* structure. The functions providing the *read* and *write* operations on the SysFS files should return the number of bytes read or written. The network devices, which are covered later, contain a *class\_device* structure. Considering the device *eth0* the new attributes would appear in */sys/class/net/eth0/myattrs/*. The *class\_device* also holds a link to a *device* structure which essentially is the the basic representation of any device. This structure maintains all the connections to busses, drivers and some very specific informations on power management, DMA and other things we usually do not want to get in touch with. This example shows how a *class\_device* can be extended by attributes. This is what we usually want, as there lies the information a user/administrator has to deal with. The attributes exported by *device* cover mostly the mentioned low-level details which are usually read-only.

Listing 4.8: Look into SysFS

```
1 /sys
2 |-- block
3     |-- ...
4 |-- bus
5     |-- ...
6     |-- pci
7         |-- devices
8             |-- ...
9             |-- 0000:00:1d.1 -> ...
10            |-- ...
11            |-- drivers
12                |-- Intel ICH
13                    |-- 0000:00:1f.5 -> ...
14                    |-- ...
15                    |-- ...
16                |-- uhci_hcd
17                    |-- 0000:00:1d.0 -> ...
18                    |-- 0000:00:1d.1 -> ...
19                    |-- ...
20            |-- ...
21            |-- usb
22                |-- devices
23                |-- drivers
24 |-- class
25     |-- input
26         |-- input0
27             |-- device -> ...
28             |-- ...
29 |-- devices
30 |-- firmware
31 |-- kernel
32 |-- module
33 |-- power
```

Listing 4.9: Adding a group of attributes

```

1  static ssize_t store_attr_a(struct class_device *cd,
2                          const char *buf,
3                          size_t len)
4  {...}
5
6  static ssize_t store_attr_b(...){}
7  static ssize_t store_attr_c(...){}
8
9  static ssize_t show_attr_c(struct class_device *cd,
10                          const char *buf,
11                          size_t len)
12 {...}
13
14 static CLASS_DEVICE_ATTR(attr_a, S_IWUSR,
15                          NULL, store_attr_a);
16 static CLASS_DEVICE_ATTR(add, S_IWUSR,
17                          NULL, store_attr_b);
18 static CLASS_DEVICE_ATTR(maxdiff, S_IWUSR | S_IRUGO,
19                          show_attr_c, store_attr_c);
20
21 static struct attribute *myattrs[] = {
22     &class_device_attr_attr_a.attr,
23     &class_device_attr_attr_b.attr,
24     &class_device_attr_attr_c.attr,
25     NULL
26 };
27
28 static struct attribute_group mygroup = {
29     .name = "myattrs",
30     .attrs = myattrs,
31 };
32
33 int add_myattrs(struct class_device *dev)
34 {
35     struct kobject *kobj = &dev.kobj;
36     int err;
37     err = sysfs_create_group(kobj, &mygroup);
38     if(err)
39     {
40         pr_info("%s can't create group %s\n",
41               __FUNCTION__, mygroup.name);
42     }
43     return err;
44 }
45

```

## 4.5 The Network subsystem

The Linux network subsystem breaks with the unix philosophy of everything being a file. Contrary to block and char devices, network devices do not have an entry point in the `/dev` directory. There is usually no reason to perform `read` or `write` operations on a network device. These operations are performed on a socket, of which many hundreds can be multiplexed to a network interface. A network interface has to provide means for transmitting and receiving packets. The network subsystem is completely independent of protocols - either hardware or software - albeit providing major support for ethernet devices and the TCP/IP protocol suite. Implementing a device similar to an ethernet device is very tempting, so that even the `plip` device, which is a network device that links two computers via their parallel ports, resembles an ethernet device in many ways.

### 4.5.1 Initialization

A network device is created using the function `alloc_netdev` and registered with the network subsystem using `register_netdev`. The usage of these functions is demonstrated in listing 4.11. The function `mydev_create` carries out all steps necessary to create a new network device. The structure type `mydev_private` is the place where device specific data is stored. The function `alloc_netdev` allocates the space for this private data, too. It is in fact appended to the structure `net_device`. The pointer `priv` links to the start of the private data. Listing 4.11 also shows how the `class_device` structure discussed earlier is used in a specific subsystem.

Listing 4.10: Initialization of a network device

```
1 struct mydev_private{
2     /* private fields */
3 };
4
5 void mydev_setup(struct net_device *dev)
6 {
7     /* custom initialization code */
8 }
9
10 struct net_device *mydev_create()
11 {
12     mydev = alloc_netdev(sizeof(struct mydev_private),
13                          "mydev%d", mydev_setup);
14     if(mydev)
15     {
16         if(register_netdev(mydev))
17         {
18             /* error handling */
19             free_netdev(mydev);
20         }
21     }
22     else
23     {
24         /* error handling */
25     }
26 }
```

Listing 4.11: Main network device infrastructure

```

1 struct net_device *alloc_netdev (
2     int sizeof_priv ,
3     const char *name,
4     void (*setup)(struct net_device *));
5 void free_netdev(struct net_device *dev);
6 int register_netdev(struct net_device *dev);
7 void unregister_netdev(struct net_device *dev);
8
9 struct net_device
10 {
11     ...
12     void *priv;
13     ...
14     struct class_device class_dev;
15     ...
16 }

```

### 4.5.2 Default interface

The network subsystem requires a device to implement a set of default functions. During initialization, the driver has to store the pointers to the implementing functions into the appropriate fields of the *net\_device* structure (cf. listing 4.10). Not all of these functions have to be implemented specifically, as the kernel includes some default implementations which are enabled through *netdev\_alloc*. A list of the most common candidates for custom implementation is given in listing 4.12.

Listing 4.12: Network device interface service routines

```

1 int (*open)(struct net_device *dev);
2 int (*stop)(struct net_device *dev);
3 int (*hard_start_xmit) (struct sk_buff *skb ,
4     struct net_device *dev);
5 int (*set_mac_address)(struct net_device *dev ,
6     void *addr);
7 int (*do_ioctl)(struct net_device *dev ,
8     struct ifreq *ifr , int cmd);
9 int (*set_config)(struct net_device *dev ,
10     struct ifmap *map);
11 int (*change_mtu)(struct net_device *dev , int new_mtu);
12 void (*tx_timeout) (struct net_device *dev);

```

- **open**  
This function is called as soon as *ifconfig* activates the device. Any resources should be initialized at this point, i.e. in a physical device this includes IRQ, DMA, et.
- **stop**  
This is the opposite to *open*.
- **hard\_start\_xmit**  
The actual workhorse of a network device which transmits packets. Will be covered in-depth in the next section.

- `set_mac_address`  
If a device is able to set its mac address, e.g. in a register of the chip on the adapter, then this function would perform this low-level work. The default implementation just sets the corresponding field `net_device->dev_addr`.
- `do_ioctl`  
Only if the interface is desired to perform specific *ioctl* operations this field must be non-null. The implementation of custom *ioctl* operations is not covered in this thesis<sup>1</sup>
- `change_mtu`  
If the MTU for this interface changes, this function is called.
- `tx_timeout`  
If a packet transmission fails to be completed within reasonable time, this function is supposed to handle the problem and to resume transmission.

### 4.5.3 Packet transport

Packet transport can be split in two parts: sending and receiving. Sending is always initiated by the network stack. The network interface gets the data to be sent via the `hard_start_xmit` function mentioned in the previous section. Reception is usually due to an interrupt caused by a packet coming over the wire and reaching the controller on the network adapter. Because of the impossibility to deal with these device-specific topics, the device-independent structure `sk_buff` which is central to packet transport will be explained.

The protocol-independency of the `sk_buff` structure in listing 4.14 is clearly visible in the excessive usage of unions. This structure perfectly fits the packet-oriented nature of most modern network protocols. It integrates the header-data for three protocol-layers, the actual payload and a lot of administrative information. The latter mostly relate to the packet filter (Netfilter) and caching. The `sk_buff` system includes several functions to manipulate this non-trivial structure. Listing 4.13 shows the signature of these functions.

Listing 4.13: `sk_buff` manipulation

```

1 struct sk_buff *skb_clone(struct sk_buff *skb,
2                          int priority);
3 struct sk_buff *alloc_skb(unsigned int size,
4                          int priority);
5 void skb_trim          (struct sk_buff *skb,
6                          unsigned int len)
7 unsigned char *skb_pull (struct sk_buff *skb,
8                          unsigned int len);
9 unsigned char *skb_push (struct sk_buff *skb,
10                          unsigned int len);
11 unsigned char *skb_put  (struct sk_buff *skb,
12                          unsigned int len);

```

- `skb_clone`  
Duplicates an `sk_buff` structure in its entirety

<sup>1</sup>With the introduction of `sysfs` the *ioctl* mechanism has become obsolete in most cases. The author considers *ioctl* harmful as every new *ioctl* operation is like a new system call. The kernel API therefore changes rapidly and becomes complex. Further, *ioctl* operations are assigned global numbers which have to be coordinated to not overlap between different devices.

- `alloc_skb`  
Allocates an *sk\_buff* structure. This is mainly used in the receiving part of a network driver.
- `skb_pull`  
Removes data from the start of the buffer. This functions returns a pointer to the next data in the buffer. Subsequent calls to `skb_push` will overwrite the old data.
- `skb_push`  
Adds data to the start of the buffer. A pointer to the new start is returned.
- `skb_put`  
Adds data to the end of the buffer. A pointer to the start of the extra data is returned.

Listing 4.14: Excerpt of the `sk_buff` structure

```

1 struct sk_buff {
2     struct sk_buff      *next;
3     struct sk_buff      *prev;
4     struct sk_buff_head *list;
5     struct sock          *sk;
6     struct timeval       stamp;
7     struct net_device    *dev;
8     struct net_device    *input_dev;
9     struct net_device    *real_dev;
10
11     union {
12         struct tcphdr    *th;
13         struct udphdr    *uh;
14         struct icmphdr   *icmph;
15         struct igmp_hdr  *igmph;
16         struct iphdr     *iph;
17         struct ipv6hdr   *ipv6h;
18         unsigned char    *raw;
19     } h;
20
21     union {
22         struct iphdr     *iph;
23         struct ipv6hdr   *ipv6h;
24         struct arphdr    *arph;
25         unsigned char    *raw;
26     } nh;
27
28     union {
29         unsigned char    *raw;
30     } mac;
31     /* ... */
32     unsigned int         len,
33                         data_len,
34                         mac_len,
35                         csum;
36     unsigned char        local_df,
37                         cloned,
38                         pkt_type,
39                         ip_summed;
40     __u32                 priority;
41     unsigned short        protocol,
42                         security;
43     /*
44     ... destination cache ...
45     ... NETFILTER ...
46     */
47     unsigned char        *head,
48                         *data,
49                         *tail,
50                         *end;
51 };

```

## Chapter 5

# Implementation of the virtual interface

The virtual network interface for transparent heterogeneous mobile ad hoc networks consists of three parts.

- A kernel module providing the actual network interface
- A library providing programmatic access to the configurable options
- A userspace utility to manage virtual interfaces

### 5.1 The kernel module

The original work [8] adapted the bridge code, whereas this work started from scratch, albeit taking a lot of inspiration out of the bridge code.

The requirements for the module were as follows.

- Provide the functionality of the original virtual interface, namely
  - Attach network devices
  - Maintain a neighbor database
  - Choose outgoing link according to priority
  - Provide some fuzziness, i.e. *maxdiff*
  - Provide a broadcast emulation
- Configuration via SysFS
- No use of custom *ioctl*s
- Implementation should be as simple as possible
- Minimize overhead

#### 5.1.1 User interface

The network interface exposes its functionality to the network subsystem via well-defined interfaces. The configuration specific to the virtual interface is only available through SysFS. The following shows which operations the user is able to perform using the files in SysFS.

- Driver
  - /sys/bus/platform/driver/vi/
  - Show version  
version
  - Create a virtual interface  
add
  - Remove a virtual interface  
remove
- Device
  - /sys/class/net/vi<x>/vi/
  - Attach physical network interface  
add
  - Detach physical network interface  
remove
  - Manipulate *maxdiff* value  
maxdiff
- Port
  - /sys/class/net/vi<x>/vi/ports/eth<y>/viport/  
/sys/class/net/eth<y>/viport/
  - Manipulate *priority*  
priority

The files may be manipulated using *cat* and *echo*.

### 5.1.2 Registering with the device model

The driver registers with the platform bus as there is no real bus it belongs to. The driver's registration is necessary because there needs to be an interface to the driver in order to instantiate a virtual interface.

### 5.1.3 The hook in the networking stack

**Listing 5.1:** Patch #2 to net/core/dev.c

```
1 if (handle_vi(&skb, &pt_prev, &ret, orig_dev))
2     goto out;
```

**Listing 5.2:** Patch to include/linux/netdevice.h

```
1     /* vi stuff */
2     struct net_vi_port *vi_port;
```

The locations to insert the patches in listings 5.3, 5.1, 5.2 are found by looking for the strings "handle\_bridge" and "br\_port" in *dev.c* and *netdevice.h*, respectively. The code is inserted just below the bridge code which looks very similar.

Listing 5.3: Patch #1 to net/core/dev.c

```
1 #if defined(CONFIG_VI) || defined (CONFIG_VI_MODULE)
2 int (*vi_handle_frame_hook)(struct net_vi_port *p,
3                             struct sk_buff **pskb);
4 struct net_vi;
5
6 static __inline__ int handle_vi(
7     struct sk_buff **pskb,
8     struct packet_type **pt_prev, int *ret,
9     struct net_device *orig_dev)
10 {
11     struct net_vi_port *port;
12
13     if ((*pskb)->pkt_type == PACKET_LOOPBACK ||
14         (port = rcu_dereference((*pskb)->dev->vi_port)) == NULL)
15         return 0;
16     return vi_handle_frame_hook(port, pskb);
17 }
18 #else
19 #define handle_vi(skb, pt_prev, ret, orig_dev) (0)
20 #endif
```

The hook is placed in the general packet reception routine of a network device. Before passing the *sk\_buff* to the upper layers it is checked if it has to be passed to a virtual interface or to the bridge.

A drawback of this approach is the necessity to patch and recompile the kernel. Different, less invasive approaches will be discussed in chapter 8.

### 5.1.4 The neighbor database

The neighbor database is a hashtable with the hash function calculated on the mac address. A doubly linked list for each hash value contains the entries corresponding to neighbors. The structure of a neighbor entry can be seen in listing 5.4.

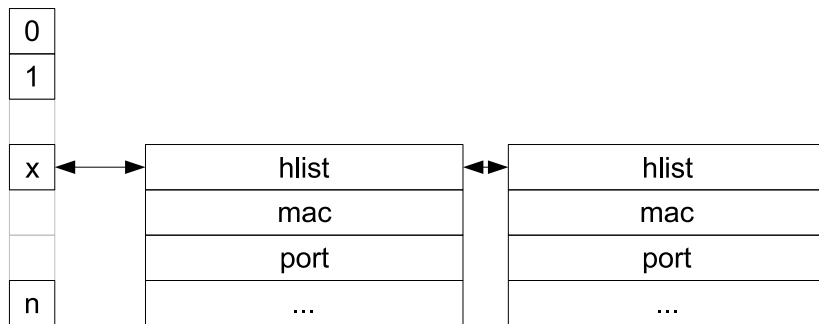
**Listing 5.4:** Structure net\_vi\_ndb\_entry

```

1 struct net_vi_ndb_entry
2 {
3     struct hlist_node      hlist;
4     atomic_t              use_count;
5     struct mac_addr       addr;
6     struct net_vi_port    *dst;
7     struct rcu_head       rcu;
8     unsigned long         ts;
9     unsigned              is_local:1;
10 };

```

**Figure 5.1:** The neighbor database (simplified)



The fields of this structure are used as follows.

- hlist  
The linked list
- use\_count  
An atomic reference counter
- addr  
The address of the neighbor. Local devices are neighbors, too.
- dst  
The port through which a neighbor is reached
- rcu  
Structure for the RCU-mechanism. This is used for adding and removing entries.
- ts  
A timestamp. The difference of such timestamps are compared with the *maxdiff value*
- is\_local  
As mentioned, local devices are neighbors, too. This field differentiates between them and real neighbors.

**Insertion** The function to insert and update entries into the neighbor database is the same. First, the hashtable is searched for a matching entry. If one is found, it is updated, otherwise a new entry is created. The update sets the timestamp to the kernel time *jiffies*.

**Outgoing link selection** Outgoing links are selected according to the algorithm in listing 5.5.

**Listing 5.5:** Link selection

```

1 list = table[hash(mac)]
2 outgoing = NULL
3 foreach(entry in list)
4 {
5     if(entry.mac == mac)
6     {
7         if(outgoing == NULL) outgoing = entry;
8         elseif(entry.port.priority < outgoing.port.priority)
9         {
10            if(outgoing.ts - entry.ts < maxdiff) outgoing = entry;
11        }
12        else
13        {
14            if(entry.ts - outgoing.ts > maxdiff) outgoing = entry;
15        }
16    }
17 }

```

The implementation of this section is found in *vi\_ndb.c*

### 5.1.5 Processing incoming packets

Incoming packets reach the virtual interface through the hook in *dev.c*.

First, the sender's entry in the neighbor database is updated or created, then the packet is passed up if:

- the virtual interface is in promiscuous mode
- the packet was sent to the broadcast address
- the destination address is the local address
- the destination address belongs to one of the ports

The implementation of this section is found in *vi\_input.c*

### 5.1.6 Processing outgoing packets

Outgoing packets reach the virtual interface through the *hard\_start\_xmit* hook of the network device default interface. A packet is sent according to the following policy:

- Broadcast packet
  - transmit over all attached interfaces
- Normal packet
  - Neighbor known
    - transmit over the corresponding outgoing link

- Neighbor unknown  
transmit over all attached interfaces

The implementation of this section is found in *vi\_device.c*.

## 5.2 The libvi library

This library provides the interface given in listing 5.6. The library uses *libsysfs* [9] to access the virtual files in the SysFS to configure the virtual interface. Before actually using the functions provided by *libvi* it has to be initialized by calling *vi\_init*.

**Listing 5.6:** Virtual interface configuration library

```
1 int vi_init(void);
2 int vi_addvi(const char *name);
3 int vi_delvi(const char *name);
4 int vi_addif(const char *vi,
5             const char *ifname);
6 int vi_delif(const char *vi,
7             const char *ifname);
8 int vi_set_portpriority(const char *ifname,
9                        unsigned long prio);
10 int vi_set_maxdiff(const char *vi,
11                   unsigned long maxdiff);
12 int vi_get_portpriority(const char *ifname,
13                       unsigned long *prio);
14 int vi_get_maxdiff(const char *ifname,
15                   unsigned long *maxdiff);
```

## 5.3 The vctl command

The *vctl* command is a command-line utility which uses *libvi* to manage virtual interfaces. If *vctl* is run without parameters it displays a help message which explains how to use it.

## Chapter 6

# Experiments and Results

The performance of the virtual interface was measured according to throughput and handover-time. The neighbor database look-up and the detour the packets have to take naturally impair throughput. Handover-time in case of a vanishing link is important as packets can get lost. There are three types of handover which have to be considered. Figure 6.1 visualizes them.

All readings were taken on Dell Latitude with integrated IEEE 802.11b adapters. Bluetooth connections were provided by various USB dongles. Linux *pand* was used to setup a *BNEP* connection.

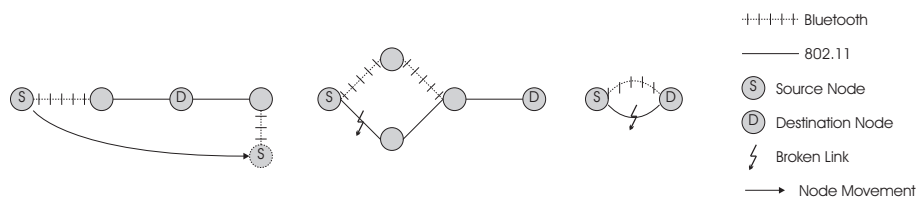
### 6.1 Throughput

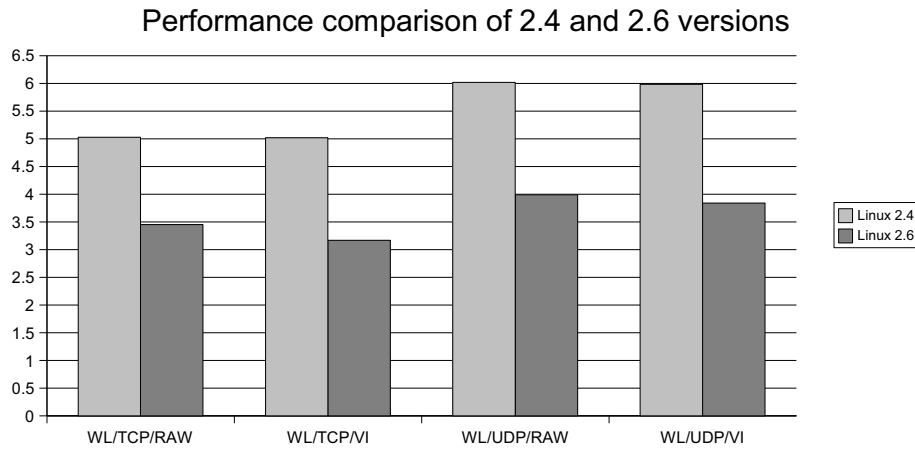
Throughput was measured using *netio*[7] which is a small program available for many platforms. It is client-server based and able to measure both TCP and UDP throughput. See the results presented in table 6.1 and 6.2. *RAW* measurements were taken without virtual interface. The varying packet sizes are important to determine the per-packet overhead. The values are averaged over 10 readings and show the throughput between two nodes in Mbps and the standard deviation  $\sigma$ .

Compared to the virtual interface of original paper the new virtual interface has a significantly larger overhead. Measurements with the Linux Ethernet Bridge on kernel 2.6. showed that it has the same problems. Throughput including a bridge is about 30% lower than raw throughput. The virtual interface can still be compared with the bridge because the neighbour database (*vi*) and the forwarding database (*bridge*) are quite similar and both use the same locking mechanisms.

Figure 6.2 shows the relation of the throughput on Linux 2.4 and Linux 2.6. Interestingly also the raw throughput on 2.6 is much lower than on 2.4 on the same hardware.

**Figure 6.1:** Horizontal, Diagonal, and Vertical handover



**Figure 6.2:** Wireless LAN throughput on Linux 2.4 and Linux 2.6**Table 6.1:** WLAN throughput in Mbps

	Packet	TCP	$\sigma$	UDP	$\sigma$
RAW	1k	3.4477	0.0221	3.5164	0.0025
	2k	3.4555	0.0250	3.4688	0.0000
	4k	3.4500	0.0234	3.8906	0.0588
	8k	3.4641	0.0066	3.8711	0.0124
	16k	3.4477	0.0377	3.8883	0.0074
	32k	3.4211	0.1119	3.9992	0.0607
VI	1k	3.1797	0.1053	3.2891	0.1148
	2k	3.1742	0.1022	3.5586	0.3257
	4k	3.1711	0.1006	3.6234	0.3131
	8k	3.1578	0.0978	3.7383	0.2988
	16k	2.7477	0.8312	3.8086	0.2374
	32k	3.1617	0.0938	3.8422	0.1498

**Table 6.2:** Bluetooth throughput in Mbps

	Packet	TCP	$\sigma$	UDP	$\sigma$
RAW	1k	0.5484	0.0911	0.5709	0.0625
	2k	0.5156	0.0772	0.6841	0.0815
	4k	0.5158	0.0822	0.6415	0.1035
	8k	0.5501	0.0789	0.6949	0.0963
	16k	0.5470	0.0768	0.7015	0.1264
	32k	0.5625	0.0746	0.7090	0.1357
VI	1k	0.5306	0.0732	0.6340	0.0684
	2k	0.5805	0.0837	0.6466	0.0516
	4k	0.5818	0.0784	0.6243	0.0923
	8k	0.5471	0.0611	0.6573	0.0896
	16k	0.5837	0.0867	0.6715	0.1359
	32k	0.5445	0.0603	0.7361	0.1479

**Table 6.3:** Handover time in seconds

Type	From	To	Routing	Interface	Time	$\sigma$
Horizontal	WL	WL	AODV	Raw	2	0
				VI	2	0
			OLSR	Raw	1.02	0.06
	BT	BT	AODV	VI	1.04	0.08
				VI	3.5	9
			OLSR	VI	8	3.01
Diagonal	WL	BT	AODV	VI/10	1.65	0.67
				VI/1000	2.05	0.55
			OLSR	VI/10	1.15	0.71
				VI/1000	7	0.62
Vertical	WL	BT	AODV	VI/10	0	0
				VI/1000	0.7	0.35
			OLSR	VI/10	0	0
				VI/1000	1.05	0.55

## 6.2 Handover

- Horizontal handover  
The MAC level protocol remains the same, but the route changes.
- Vertical handover  
The neighbour remains the same, but the MAC level protocol changes.
- Diagonal handover  
The MAC level protocol and the route change simultaneously.

The handover times were measured using *ping*. The number of missing packets were counted and multiplied by the ping frequency. The results are presented in table 6.3. Handover times are averages over 10 readings with standard deviation  $\sigma$ . Entries of the form "VI/x" must be understood as "Virtual Interface with a maxdiff value of x". The large standard deviations in the horizontal Bluetooth handover are caused by the Bluetooth inquiry mechanism.

The handover times from Bluetooth to WLAN, either diagonal or vertical, are similar to the times from WLAN to Bluetooth.

The handover times of the original and the new virtual interface do not differ as much as the throughput, i.e. they are essentially equal.



# Chapter 7

## Users' guide

### 7.1 Installation

#### Linux kernel

- Install the Linux 2.6.12 sources
- Configure the kernel to your needs
- Build and install the kernel
- Reboot and verify
- Apply the patch *vimodule.patch*
- Rebuild and install the kernel

#### Routing protocols

- Get the source distribution of AODV-UU [1]
- Install according to the AODV-UU installation manual
- Get the source distribution of OLSRD [6]
- Install according to the OLSRD installation manual

#### Libsysfs

- Install the development package of libsysfs for your distribution or get and install the source distribution from [9]

#### Virtual interface

- If the file *configure* does not exist in the root of the vi distribution, issue the following commands
  - *aclocal*
  - *autoheader*
  - *autoconf*
- run *./configure*
- run *make*
- run *make install*

## 7.2 Usage

The *victl* is self-explanatory. The following is an example on how to install a virtual interface.

- Load the module *vi.ko*
- Add a virtual interface using *victl addvi vi0*
- Start the interface by *ifconfig vi0 up*
- Add an existing network interface to the virtual interface by *victl addif vi0 eth1*
- Set the priority by *victl setportprio vi0 eth1 10*
- Add another existing interface by *victl addif vi0 bnep0*
- Set a MAC address for the virtual interface by *ifconfig vi0 hw ether \$MAC*
- Set an IP address by *ifconfig vi0 \$IP*

## Chapter 8

# Conclusions and Future Work

Using a virtual interface for transparent heterogeneous mobile ad hoc networks has once again proven to be a good approach. Using either *OLSR* or *AODV* reasonable handover times can be achieved, although Bluetooth connection setup may drastically impair performance. The throughput rate are little below the raw rate.

Placing a hook into *dev.c* has shown to be very invasive and not at all flexible. It is indeed not too easy to find a way to get the raw ethernet frames with a reasonable overhead. The most promising solution might be a custom Netfilter target. Such a target can be loaded and unloaded from kernel at any time. A well-understood architecture in the kernel and a userspace utility make Netfilter a powerful tool. The Netfilter target for the virtual interface and other known Netfilter targets could be combined in any favored way

Another approach might be to use the *ip\_queue* Netfilter target which allows the implementation of packet filters in userspace. Due to the necessary context switches, this solution should not be considered a real alternative.



# Bibliography

- [1] Uppsala University CoRe Group. Aodv-uu. <http://core.it.uu.se/AdHoc/AodvUUImpl>.
- [2] Free Software Foundation. Gnu general public license. <http://www.gnu.org/licenses/gpl.html>.
- [3] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 3rd edition, 2005.
- [4] Eva-Katharina Kunst Jürgen Quade. *Linux Treiber entwickeln*. dpunkt.verlag GmbH, 1st edition, 2004.
- [5] Cross-Referencing Linux. Lxr. <http://lxr.linux.no>.
- [6] OLSRD Project. Olsr daemon. <http://www.olsr.org>.
- [7] Kai Uwe Rommel. Netio. <http://www.ars.de/ars/ars.nsf/docs/netio>.
- [8] Patrick Stuedi and Gustavo Alonso. Transparent heterogeneous mobile ad hoc networks. In *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services. MobiQuitous, San Diego*, pages 237–246, 2005.
- [9] Linux Diagnostic Tools. sysfsutils. <http://linux-diag.sourceforge.net/Sysfsutils.html>.