Horizontal Heterogeneity through SecMIP controlled infrastructure

COMPUTER SCIENCE PROJECT

from

Simi Winiker

Computer Networks and Distributed Systems (RVS)
University of Bern

Professor: Prof. Dr. Torsten Braun

Bern, December 2001

ABSTRACT

Since a few years almost every enterprise has build its own intranet to host business related info, to provide a window where employees or work groups can present their work and after all to offer the worker the possibility to store his documents on a server to avoid the loss of data or to save disk space on his own workstation.

On the other hand we could observe that more and more network accessing devices sold in the last years are mobile stations like laptops or handhelds. Not only the employees have to be flexible but also the technologies and they should not limit the possibilities of networking in a moving scenario.

Also in the third generation of mobile telephony (UMTS) it will be an important task to have access to the internet and to business related data on the enterprise's own intranet.

This evolution in the need of mobility has showed the limitations of the current TCP/IP Protocol stack which has become the most important protocol stack. This problems are treated in the new version of this protocol stack (IPv6) and in the current version (IPv4) Mobile IP has solved the IP related aspects in this topic.

One of the biggest problems of Mobile IP is its lack of security. For this reason several efforts have been made in the last few years, also in the University of Berne. There has been developed a variant of Mobile IP with an integration of IPSec to solve the security problems of Mobile IP. This solution is called SecMIP (Secured Mobile IP [2]).

With help of this solution it is possible to have a secured access on private data from everywhere. This solution solves on the one hand the problem of changing IP addresses when moving and on the other hand the security problems when leaving the secured home network.

These suggestions are on the IP layer and do not cover the problems of the lower layers of the TCP/IP protocol stack. When we are changing from one network to another often not only the IP layer has to be modified but also the link layer. E.g. when we are plugged to an Ethernet network and decide to move and want to use Wireless LAN meanwhile. So a handover mechanism has to be provided in order to offer the full Portable Office solution.

TABLE OF CONTENTS

1 INTRODUCTION 1
1.1 THE PURPOSE OF A HORIZONTAL HETEROGENEITY 1
1.2 ABOUT THIS PROJECT 1
1.3 THE STRUCTURE OF THIS DOCUMENT 2
1.3.1 CARD CONFIG TOOL 2
1.3.2 HADES 2
1.3.3 JORDI 3
2 THE CURRENT SITUATION 4
2.1 MOBILE IP (MIP) 4
2.1.1 MOBILE IP FUNCTIONAL ENTITIES 4
2.1.2 HOW MOBILE IP WORKS 4
2.2 IPSEC 6
2.2.1 WHERE DOES IPSEC FIT IN? 6
2.2.2 SECURING THE IP Layer 6
2.2.3 SECURITY SERVICES OFFERED BY IPSEC 7
2.2.4 IPSEC PROTOCOL SUITE 7
2.3 SECURED MOBILE IP (SECMIP) 8
2.3.1 THE PROTOTYPE 8
2.3.2 SECURE AND MOBILE INTERNETWORKING 8
2.3.3 SECMIP A SECUREDMOBILE IP IMPLEMENTATION 10
2.4 ENHANCEMENT TO SECMIP 17
2.4.1 SECURED MOBILE IP ON HETEROGENEOUS NETWORKS 17
2.4.2 HOW DOES IT WORK? 17
2.4.3 SECMIP MODULE 18
2.4.4 CARDMANAGER MODULE 19
2.4.5 HARDWARE INDEPENDENT SECMIP DEVICE 20
2.4.6 WHERE DOES HADES FIT IN? 20
3 THE CARD CONFIG TOOL 22
3 THE CARD CONFIG TOOL 23
3.1 REQUIREMENT SPECIFICATION 23
3.2 DESIGN 25
3.2.1 THE PCMCIA PROCESS IN LINUX 25
3.2.2 CARD CONFIG TOOL AND PCMCIA PROCESS INTERACTION 28
3.2.3 THE BASIC STRUCTURE OF THE CARD CONFIG TOOL 30
3.2.4 CLASS STRUCTURE OF THE CARD CONFIG TOOL 32
3.2.5 EXPLANATIONS OF THE CHOSEN DESIGN 34
3.3 IMPLEMENTATION 36
3.3.1 CLASS DIAGRAMS OF THIS IMPLEMENTATION 37
3.3.2 ABERRATIONS FROM THE DESIGN 39
3.4 DEVELOPING TECHNOLOGIES USED 40
3.4.1 GNU/LINUX 40
3.4.2 PYTHON 41
3.4.3 GTK 43
3.4.4 GLADE 44
3.4.5 GLADE-PYTHON CODE GENERATOR 44
3.5 LAST BUT NOT LEAST: HOW DOES IT LOOK LIKE? 45
4 HADES 47
4.1 HADES: HARDWARE INDEPENDENT SECMIP DEVICE 47

4.2 THE CONCEPT 47
4.3 WHERE HADES FITS IN 47
4.4 A FEW WORDS ABOUT THE LINUX KERNEL 50
4.4.1 PARTS OF THE KERNEL 50
4.4.2 NETWORK INTERFACES 52
4.4.3 NETWORK DRIVERS 52
4.5 IMPLEMENTATION DETAILS: THE FIRST APPROCH 53
4.5.1 VIRTUAL NETWORK INTERFACES 53
4.5.2 HADES CODE 53
4.6 USING HADES 61
4.7 TESTS 63
4.8 SUMMARIZING 65
5 JORDI 66
5.1 OVERVIEW 66
5.2 SUPERVISING THE LINK LAYER 67
5.2.1 ETHERNET 67
5.2.2 WIRELESS LAN 67
5.2.3 CONNECTION ORIENTED DEVICES 69
5.3 CHANGING THE CONNECTION 70
5.4 UPDATING HADES AND INFORMING SECMIP 72
5.5 CONCLUSIONS 72
6 OUTLOOK 73
6.1 CARD CONFIG TOOL 73
6.1.1 SCHEMES 73
6.1.2 INSTALLATION 73
6.1.3 THE GUI 73
6.1.4 NEW TECHNOLOGIES 74
6.1.5 NON PCMCIA DEVICES 74
6.2 HADES 74
6.3 JORDI 76
7 RELATED WORK 77
7.1 FUZZY LOGIC FOR A HETEROGENEOUS IP ENVIRONMENT 77
7.1.1 INTRODUCTION 77
7.1.2 HANDOVER MANAGEMENT 77
7.1.3 APPLICATION OF THE FUZZY-LOGIC-BASED HANDOVER CONCEPT 78
8 ACKNOWLEDGEMENTS 79
9 REFERENCES 80

1 INTRODUCTION

1.1 THE PURPOSE OF A HORIZONTAL HETEROGENEITY

In the last years mobility has become one of the most important issues in computer sciences and telecommunications. People like to access their data from everywhere. This has lead to many technologies like Wireless LAN or the GSM network and the future will bring us even more like GPRS, UMTS and Bluetooth.

The main problem of this heterogeneous scenario is the change from one network to another. When you change the network, you also have to change the configuration of your network device or even the network device itself. Mobile IP is an solution for the IP related problems of this changes and it is based on the idea of VPNs. With Mobile IP you can move from one network to another without changing your IP address.

But Mobile IP has a lack of security. For this reason SecMIP [2] has been developed. It is an integration of the IPSec solution into Mobile IP in order the get secure access to private data from unsecured networks.

But this still does not solve the problem of the handover from one network technology to another. Some handover mechanisms must be developed to provide the needed flexibility.

So, the combination of SecMIP and a handover mechanism that is transparent to the upper layers of the TCP/IP stack would lead to a scenario, where we can move along different networks without losing connection.

1.2 ABOUT THIS PROJECT

The work on this project started in February 2001. The requirements it should achieve are based on a patent. This patent is shown in chapter 2.4. The whole project where this work fits in is called The Portable Office.

After studying the requirements the project was split in three parts:

- The Card Config Tool
- HadeS

- JorDI

1.3 THE STRUCTURE OF THIS DOCUMENT

The first chapter contains the schedule and the initial decisions that were made at the beginning of this project. It also gives a brief overview how the responsibilities of this project are distributed. And finally it is the part you are about to read.

The second chapter of this document concerns the work that was already done. It briefly explains the SecMIP implementation by introducing Mobile IP and IPSec and the merge of it. It also shows the already mentioned document upon which this work is based.

This project is split in three different parts. In the chapters 3, 4 and 5 the results that have been reached are explained.

Following a brief overview what the responsibilities of this different parts are:

1.3.1 CARD CONFIG TOOL

Before performing a handover it is important to make sure that a successful change on the link layer of the TCP/IP stack will lead to a successful configuration of the network interface on the IP layer of the TCP/IP stack. That fore the interfaces must have an IP configuration which will be working to prevent a handover on an interface that will not be properly brought up on the IP layer. This initial setup must be stored for every network interface that will be used before running the Portable Office implementation.

1.3.2 HADES

Hardware in**de**pendent **S**ecMIP device

This is a virtual interface. It should offer SecMIP the possibility to be based always on the same interface and to leave all the handover decisions to the Card Manager mentioned in chapter 2.4. This virtual device will make all the switches of cards and changes of configurations transparent for the SecMIP implementation.

1.3.3 JORDI

Jump **o**n **r**ight **D**evice **I**ntelligence

This part of the project is responsible to detect link layer events and to decide how to react on them. Based on the interfaces' configurations that are initially made with the Card Config Tool and on the state of the link layer, JorDI should be able to make a mature decision about which device it has to chose and bring it up. Once a proper working state has been reached JorDI informs SecMIP about the new IP and forces a handover. Unfortunately JorDI has never reached the state of a working prototype. But there are already some suggestions from our side about a possible implementation.

2 THE CURRENT SITUATION

2.1 MOBILE IP (MIP)

[1]

2.1.1 MOBILE IP FUNCTIONAL ENTITIES

Mobile IP needs three functional entities where its mobility protocol must be implemented:

Mobile Node (MN): A host or router, which can change its point of attachment from one network or sub network to another. This change of location may not concern its (home) IP address. All ongoing communications can be maintained without any interrupt.

Home Agent (HA): A router on the mobile node's home network that redirects any IP packets for the mobile node to its current location.

Foreign Agent (FA): A router on a visited network providing routing services to the MN.

2.1.2 HOW MOBILE IP WORKS

IP routes packets from a source endpoint to a destination by allowing routers to forward packets from incoming network interfaces to outbound interfaces according to routing tables. The routing tables typically maintain the next-hop (outbound interface) information for each destination IP address, according to the number of networks to which that IP address is connected. The network number is derived from the IP address by masking off some of the low-order bits. Thus, the IP address typically carries with it information that specifies the IP node's point of attachment.

To maintain existing transport-layer connections as the mobile node moves from place to place, it must keep its IP address the same. In TCP (which accounts for the overwhelming majority of Internet connections), connections are indexed by a quadruplet that contains the IP addresses and port numbers of both connection endpoints. Changing any of these four numbers will cause the connection to be disrupted and lost. On the other hand, correct delivery of packets to the mobile node's current point of attachment depends on the network number contained within the mobile node's IP address, which changes at new points of attachment. To change the routing requires a new IP address associated with the new point of attachment.

Mobile IP has been designed to solve this problem by allowing the mobile node to use two IP addresses. In Mobile IP, the home address is static and is used, for instance, to identify TCP connections. The care-of address changes at each new point of attachment and can be thought of

as the mobile node's topologically significant address; it indicates the network number and thus identifies the mobile node's point of attachment with respect to the network topology. The home address makes it appear that the mobile node is continually able to receive data on its home network, where Mobile IP requires the existence of a network node known as the home agent. Whenever the mobile node is not attached to its home network (and is therefore attached to what is termed a foreign network), the home agent gets all the packets destined for the mobile node and arranges to deliver them to the mobile node's current point of attachment.

Whenever the mobile node moves, it registers its new care-of address with its home agent. To get a packet to a mobile node from its home network, the home agent delivers the packet from the home network to the care-of address. The further delivery requires that the packet be modified so that the care-of address appears as the destination IP address. This modification can be understood as a packet transformation or, more specifically, a redirection. When the packet arrives at the care-of address, the reverse transformation is applied so that the packet once again appears to have the mobile node's home address as the destination IP address. When the packet arrives at the mobile node, addressed to the home address, it will be processed properly by TCP or whatever higher level protocol logically receives it from the mobile node's IP processing layer (layer 3).

In Mobile IP the home agent redirects packets from the home network to the care-of address by constructing a new IP header that contains the mobile node's care-of address as the destination IP address. This new header then protects or encapsulates the original packet, causing the mobile node's home address to have no effect on the encapsulated packet's routing until it arrives at the care-of address. Such encapsulation is also called tunneling, which suggests that the packet burrows through the Internet, bypassing the usual effects of IP routing.

Mobile IP, then, is best understood as the cooperation of three separable mechanisms:

- Discovering the care-of address;
- Registering the care-of address;
- Tunneling to the care-of address.

2.2 IPSEC

[1]

2.2.1 WHERE DOES IPSEC FIT IN?

Technologies securing Internet communications exist, but most of them are dedicated to specific software applications, for example PGP for mail encryption and browser-based authentication, and encryption between the browser and the web server (SSL) to protect sensitive web traffic. These restrictions do not get along with the requests of a large enterprise and the average Internet service provider that may never know precisely what applications may be running tomorrow over the networks they are building today.

The most effective way to fulfill such an application independent security is to place all security mechanisms below the application layer, i.e. in the network layer.

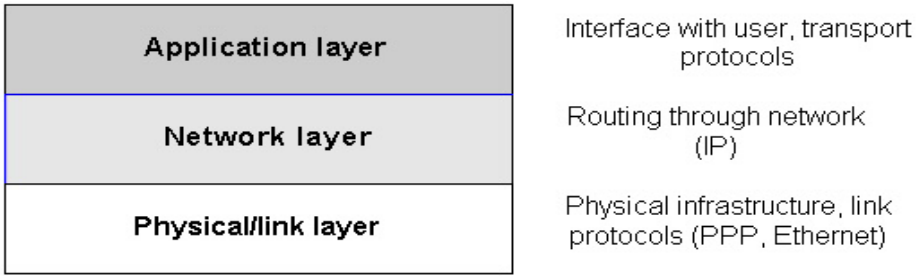


Figure 2.8 Network layers

2.2.2 SECURING THE IP Layer

An IETF working group has developed a method to secure the IP layer. They call it the IP security protocol suite (IPSec). This protocol suite adds security services to the IP layer keeping compatible with IP standard (IPv4).

Using IPSec where normally using IP, all communications for all applications and all users can be secured in a transparent way.

IPSec eases building secure virtual private networks (VPN) – a secure, private network that is as safe or safer than an isolated office LAN, but built on an unsecured, public network. Because the IPSec protocol suite is compatible with the normal IP protocol, it is enough to support IPSec on the end systems. The rest of the network in between can work just as it works now.

IPSec promises to become the new international standard, allowing different networks around the world to interconnect and communicate securely. It also promises a very good scalability while providing quiet and reliable security services.

2.2.3 SECURITY SERVICES OFFERED BY IPSEC

Three things must be ensured for security in a network environment:

- The communicating persons must really be the persons they seem to be (authentication)
- No one shall be able to eavesdrop on communications (confidentiality, privacy)
- The received communication must not be altered in any way during transmission (integrity)

2.2.4 IPSEC PROTOCOL SUITE

The IPSec-Protocol-Suite consists of three main parts:

- Authentication Header (AH) - ties data in each packet to a verifiable signature that allows to verify both the identity of the person sending data and that data has not been modified.
- Encapsulating Security Payload (ESP) – encrypts data (and even certain sensitive IP addresses) in each packet – so a sniffer somewhere on the network doesn't get anything usable.
- Internet Key Exchange (IKE) – a powerful, flexible negotiation protocol that allows users to agree on authentication methods, encryption methods, the keys to use, how long to use the keys before changing them, and that allows smart, secure key exchange.

With these three protocols it is possible to secure a connection on the IP level. After having exchanged all parameters with IKE, all packets are being encrypted and authenticated with AH/ESP.

2.3 SECURED MOBILE IP (SECMIP)

[2]

2.3.1 THE PROTOTYPE

Mobile IP needs IPSec by default as all packets between a remote MN to/from the home network should be made secure. At the time this document was written, there were no standards for this merging. It seams that the work in progress focuses rather to the integration of security in the Mobile IP version 6. Nevertheless, mobility in IP version 4 has to be evaluated, because it will take some years to introduce the new IP version to the whole Internet community. In the meantime nobody wants to work without mobility.

2.3.2 SECURE AND MOBILE INTERNETWORKING

Vipul Gupta and Gabriel Montenegro from Sun Microsystems[3] describe enhancements that enable Mobile IP operation in a network which is protected by some combination of source-filtering routers, sophisticated firewalls, and private address space. These enhancements should allow a mobile user, in the public Internet, to maintain a secure virtual presence within his firewall protected office network. This constitutes what we call a Mobile Virtual Private Network (MVPN).

The first part of their paper describes the deployment of a security architecture which allows a separation of an organization's network into two sub networks with different security guidelines. The interior network is insulated from the Internet by a perimeter network (also known as DeMilitarized Zone (DMZ)) and two packet filtering routers R1 (exterior or access router) and R2 (interior or choke router).

As an additional security measure internal network topology is hidden using application relays and private address space. Outside routers are unaware of internal addresses and inside routers (such as R2) are unaware of outside addresses. Inside routers, however, are aware of addresses on the perimeter network. All routers drop packets with an unknown destination address.

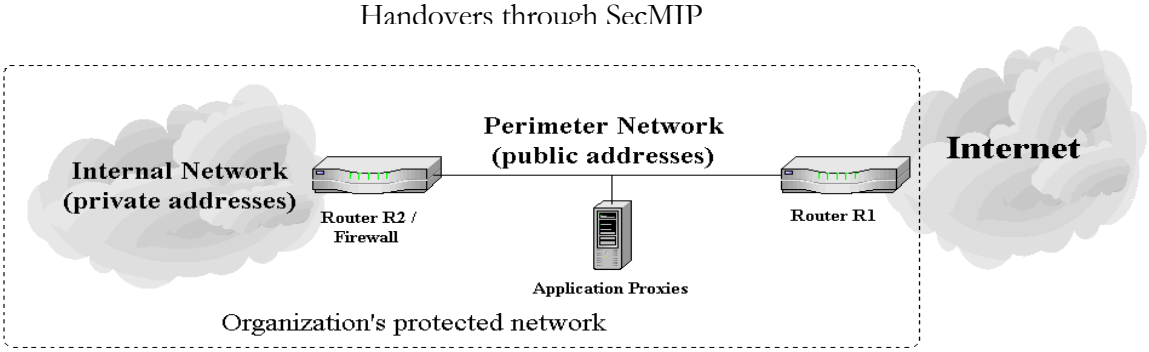


Figure 2.14 DMZ

In the proposed deployment of a secure Mobile IP, the mobile nodes work in MN decapsulation mode with collocated COA. Two tunnels are used to pass the perimeter network (i.e. MIP registration messages). The tunnel between the home agent and the firewall hides the care-of address from inside routers and that between the firewall and the mobile node hides the home agent address from outside routers. The second tunnel can be used to provide encryption and authentication (see Figure 2.15)

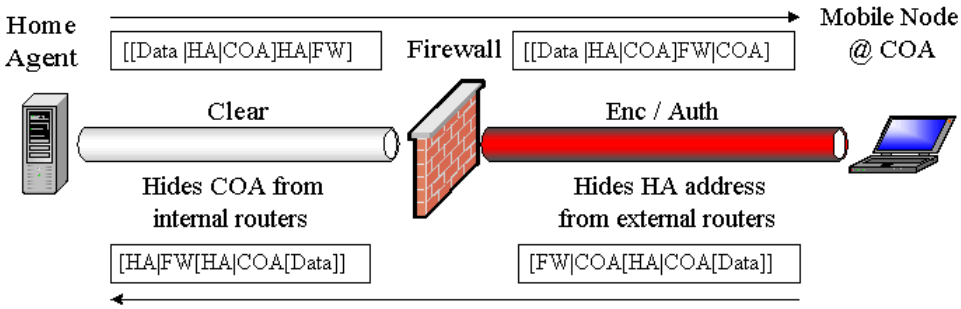


Figure 2.15 Tunnels for firewall traversal

2.3.3 SECMIP A SECUREDMOBILE IP IMPLEMENTATION

After knowing the most important proposals how to secure Mobile IP with the help of strong authentication and strong encryption, the idea was to design a new deployment architecture taking the best out of the existing concepts. The result was called SecMIP, which stands for Secured Mobile IP.

2.3.3.1 DEPLOYMENT ARCHITECTURE

Like V.Gupta and G.Montenegro proposed in their paper "secure and mobile networking"[3], a so-called screened-subnet firewall architecture was chosen, where the organization's interior network was isolated from the Internet by a DMZ. This particular architecture was chosen for its popularity and superior security characteristics. The firewall between the DMZ and the private interior network was the only entry point to the organization's private network.

Later in this text, it will be shown that this simplifies the security management significantly, since all traffic must pass this firewall, Mobile IP included. As an additional security measure, private addresses are used for the internal network. This hides the topology of the private part of the network, even if packets are tunneled (i.e. with IPSec in tunnel mode) through a public network. This private network is then called a virtual private network (VPN). To ensure privacy of such VPNs, normally encryption mechanisms are deployed to the hide the tunneled data passing in-secure parts of the Internet.

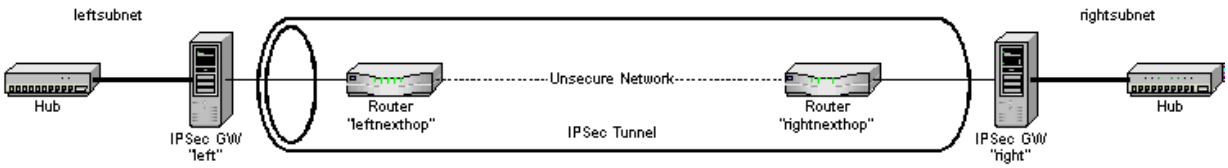


Figure 2.16 VPN

The main requirement driving our deployment of Mobile IP, and the topological placement of home and foreign agents with respect to the firewall is that the corporate network must not be exposed to any new security threats. The easiest and most effective way to fulfill this requirement is to place all Mobile IP devices (except own HAs) outside the private network, i.e. placing them in the DMZ. Of course the mobile node itself should be protected from attacks coming from other Internet nodes. This can be done with a firewall software on the MN.

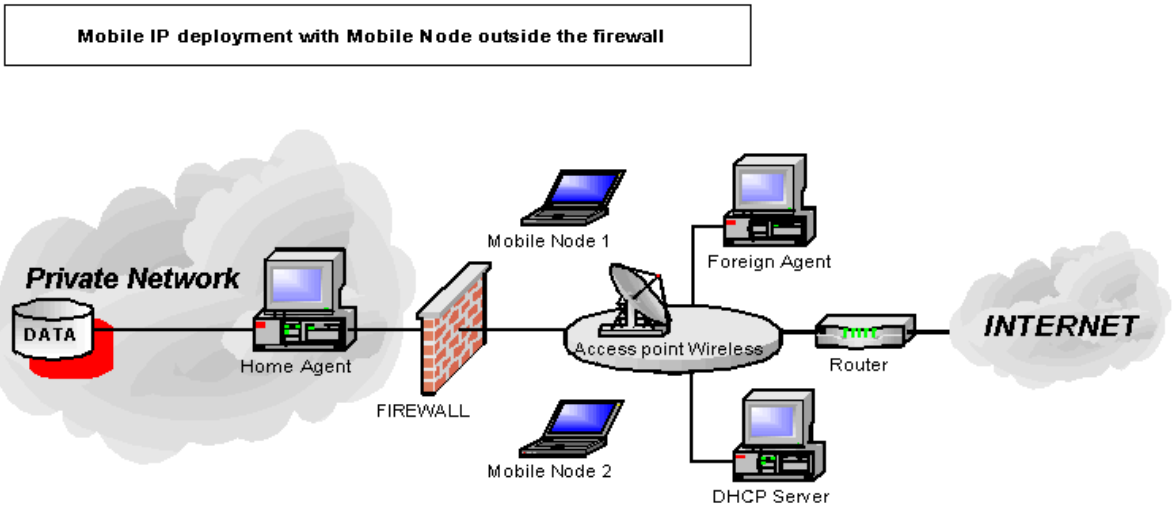


Figure 2.17 MIP device deployment

In the mobile node decapsulation mode, the mobile nodes receive their IP addresses from a DHCP server as described above. All mobile nodes are always outside the firewall, i.e. in the DMZ, even those owned by the corporation. This means that the mobile nodes are never located in the same subnetwork as the HA, while using wireless LAN and thus also never registered at home. Mobile nodes that become attached to the physically secured wired network inside the firewall, stop Mobile IP tunneling. Despite of speed reductions resulting from authenticating and encrypting data being sent from the organizations own DMZ to the firewall, the benefits concerning the security of the VPN justify this concept. It is even possible to reduce the traffic on the interior private network, as the home agent advertisements would not be needed regularly. If the organization's security policies allow the own mobile nodes to be attached inside the interior network, all Mobile IP functionalities

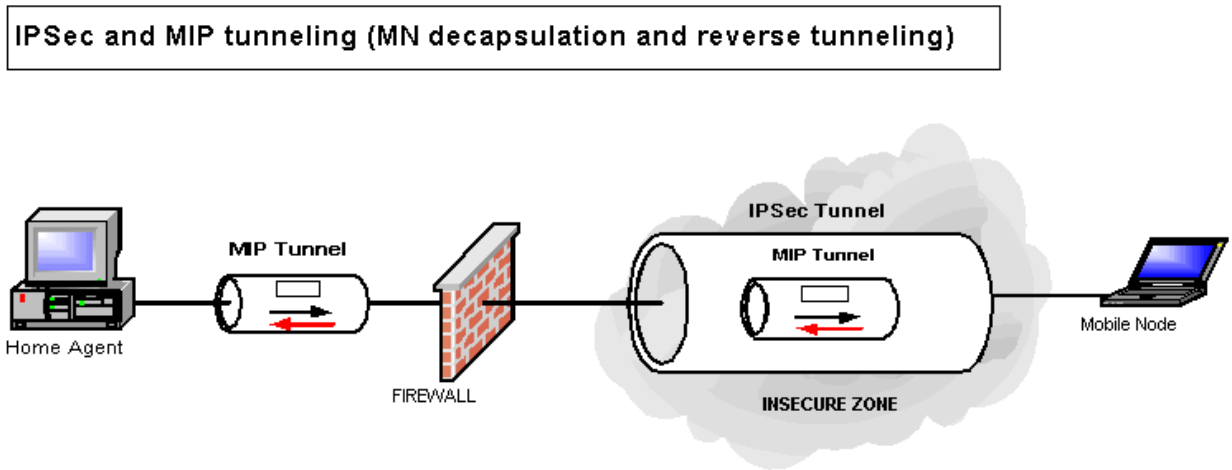
should be disabled. This allows to ensure that wireless attachment points are only used with a secured Mobile IP.

2.3.3.2 IPSEC IN SECMIP

As the mobile nodes that belong to the corporation have to traverse the firewall to access the VPN, they have to authenticate themselves to the firewall. This authentication is realized with IPSec. Since there is a real end-to-end authentication between the corporation's own mobile nodes and the firewall, they can easily be configured with a shared secret or even RSA keys. The establishment of a secure IPSec tunnel between the mobile node and the firewall makes it possible to use a lightweight (without security mechanisms) implementation of Mobile IP, since all packets traversing the public network are encrypted and authenticated by IPSec. We will discuss the security mechanisms added by IPSec for each phase of Mobile IP as registration, tunneling, re-registration later in this document.

Similar to the tunneling proposed by Sun, SecMIP uses an IPSec tunnel to protect the Mobile IP tunnel passing the insecure parts of the Internet. Within the private network, however, the Mobile IP tunnel is sufficient.

Figure 2.18 SecMIP tunneling



2.3.3.3 SECMIP OPERATION

In this section the operation mode of SecMIP is regarded in detail. This is done by going step by step with a mobile node which changes its point of attachment.

1. Network detection

Entering a conference room, a mobile node has to be connected either to a wireless network (with access point—see Figure 2.19), or to a conventional network medium like Ethernet. On this demilitarized network, foreign agent advertisements are broadcasted regularly. Catching such an ICMP message, a MN learns that it just has entered a new network. The mobile node can also send an agent solicitation to provoke an agent advertisement. Then, the MN stops the old IPSec tunnel, which was been established from an older location (with an old collocated care-of address).

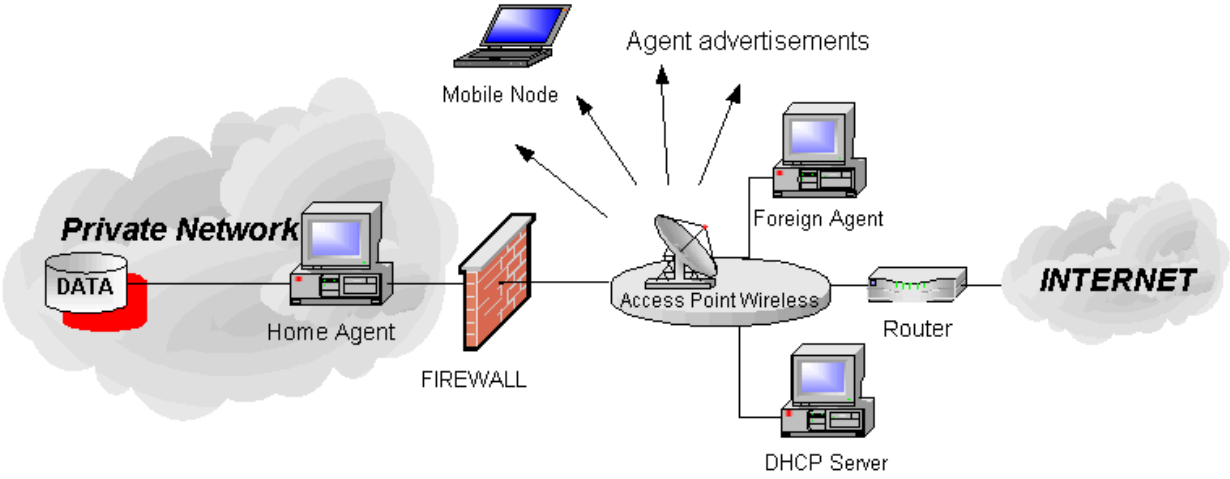


Figure 2.19 SecMIP network detection

2. Acquiring a routable IP address

In MN decapsulation mode, the mobile node needs to acquire a collocated care of address through a DHCP server. It is also possible to read available collocated care-of addresses from the foreign agent advertisements. But in this case the foreign agent has to keep track on the used addresses.

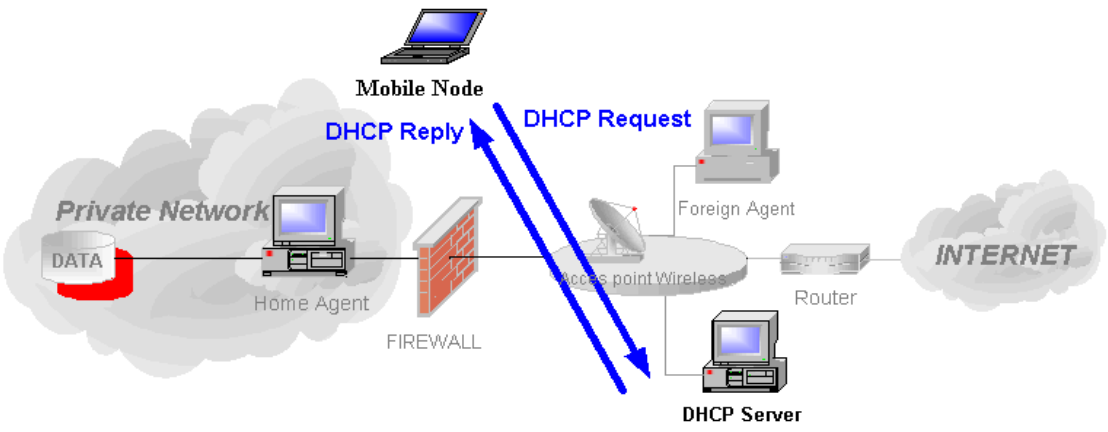


Figure 2.20 SecMIP acquiring a collocated COA

3. Establishment of a bi-directional IPSec tunnel between MN and Home Firewall

As shown in Figure 2.21, data packets pass an insecure, public network between MN and Home-Firewall. So it is by far the best approach to establish an IPSec tunnel between the MN's COA and the home firewall before any Mobile IP messages are exchanged between these two entities. The IPSec tunnel ensures authentication, integrity and privacy of every IP packet sent by the Mobile IP registration procedure.

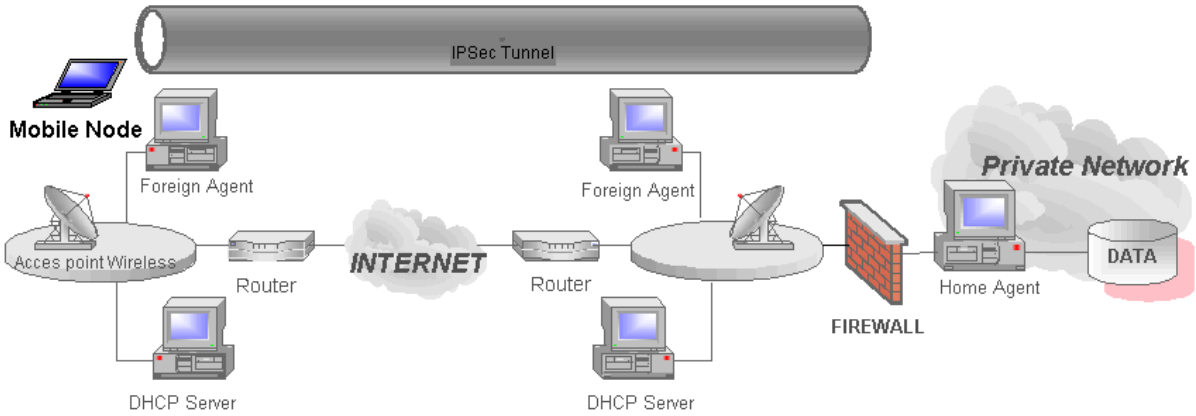


Figure 2.21 IPSec tunnel MN home firewall

4. Home Agent and MN negotiation: Mobile IP registration (light)

Since all Mobile IP negotiation between HA and MN pass the IPSec tunnel to the home site's firewall, the registration messages needs neither to be authenticated nor encrypted by the Mobile IP protocol. The security in the private network behind the firewall is supposed to be ensured. So the whole MIP registration proceeds in a secure way.

5. Data transfer from the MN to the whole Internet including its home network

Until the next movement, MN can communicate with any other CN. If the mobile node changes its location, the procedure begins at step 1.

2.3.3.4 EXCHANGED MESSAGES

The next figure shows which messages are exchanged between the involved SecMIP entities during the described steps of SecMIP.

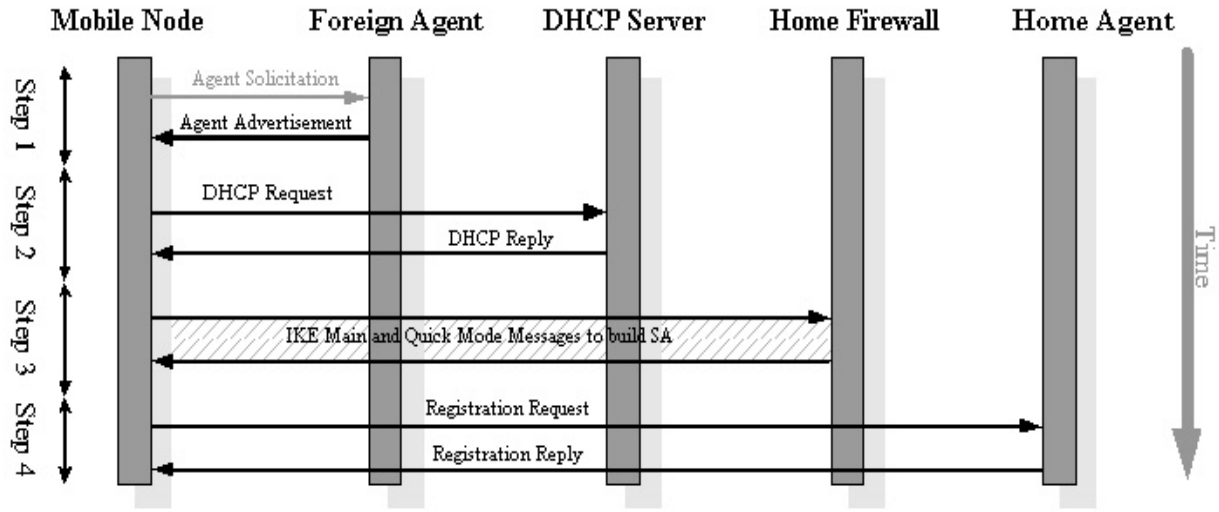


Figure 2.22 Exchanged messages

2.3.3.5 SECMIP PACKETS

The figure below shows how the exchanged IP packets are built. The packets exchanged in the first two steps are not really important for the understanding of SecMIP and therefore not described explicitly. (For detailed information please consult RFC 2002[4] for structure of Agent Advertisements and RFC 2131[5] for the DHCP). In step three, IPSec packets are exchanged between the mobile node's collocated care-of address and the Home Firewall (HF). These packets carry the information for the main and quick mode of IKE in the payload (As described in 2.2.9 Internet Key Exchange (IKE)).

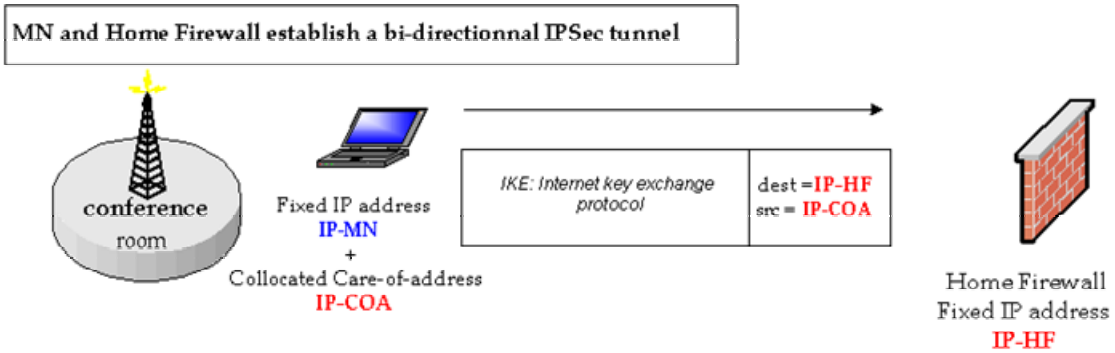


Figure 2.23 IPSec packets

After having set up a secure channel with the IPSec protocol suite, Mobile IP packets can pass the insecure Internet. These encrypted and authenticated Mobile IP packets are decrypted and decapsulated by the Home Firewall and delivered to the HA. The home agent finally decapsulates these Mobile IP packets and delivers them to the appropriate receivers, the correspondent nodes.

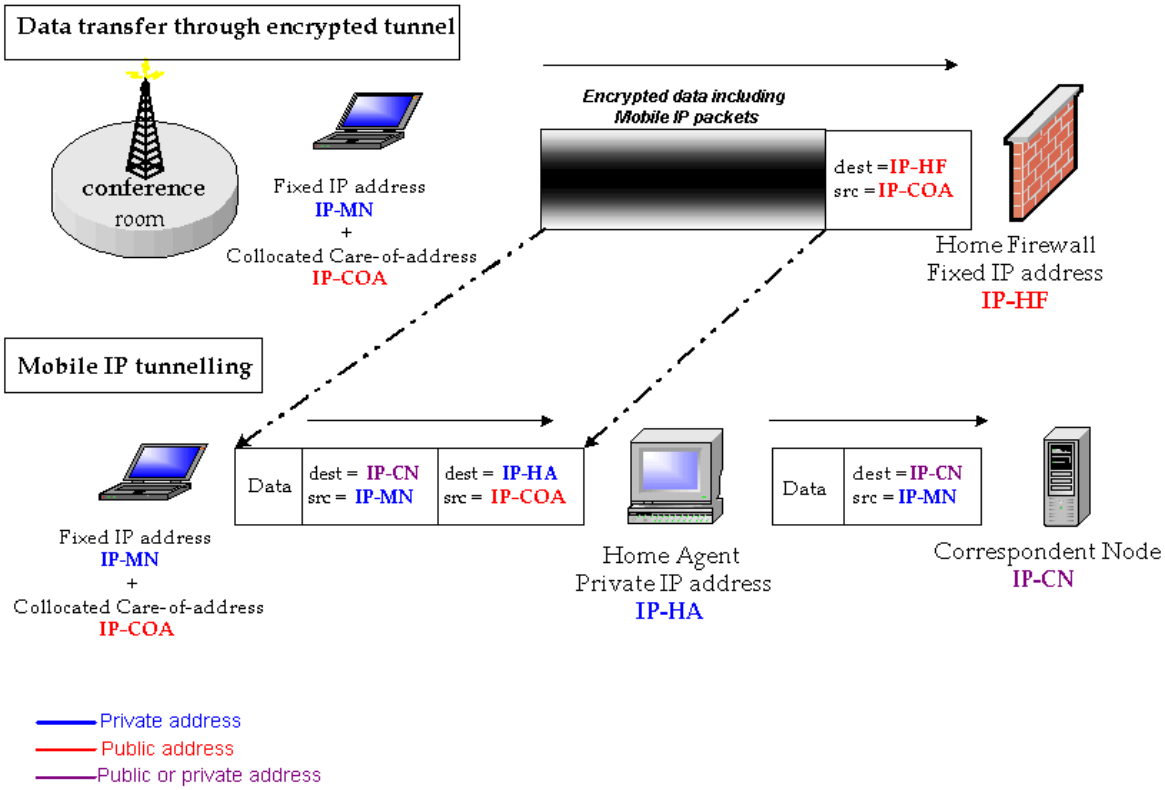


Figure 2.24 IPSec packets

2.4 ENHANCEMENT TO SECMIP

[6]

2.4.1 SECURED MOBILE IP ON HETEROGENEOUS NETWORKS

This mobility concept makes it possible to be always connected to the company's home network, independent of the used network medium. The user just decides which network devices he wants to allow the mobile node to use and portable office mechanisms do manage the connections. Using any kind of networks this concept provides a secure mobile communication to the office network.

2.4.2 HOW DOES IT WORK?

The portable office concept is based on two main middleware modules acting at different layers. First of all, Mobile IP is used, which is able to keep the mobile node's movements to other IP sub networks transparent for all layers above the network layer. The main task of Mobile IP is to authenticate a mobile node and redirect all IP packets that belong to it. Combining the security features of IPSec with the mobility management of Mobile IP, it is possible to offer a secure mobility using the insecure public Internet (SecMIP Module).

Secondly, as mobile IP does not care what connection type is offered by the link (or physical layer), there may be mobility over heterogeneous networks. The middleware module called Cardmanager that manages the communication devices works transparently for the network layer. So TCP/IP connections keep alive even if the communication medium has been changed. Being modular, this mobility concept is kept open for new communication technologies like Wireless LANs, Bluetooth, GPRS or UMTS.

The cardmanager provides a virtual network device (called HadeS – Hardware Independent SecMIP device) that is linked with the actual physical network device. This virtual device HadeS just changes its IP configuration when being linked to another physical network device. The SecMIP module manages the IPSec and Mobile IP tunnels to work with this virtual device.

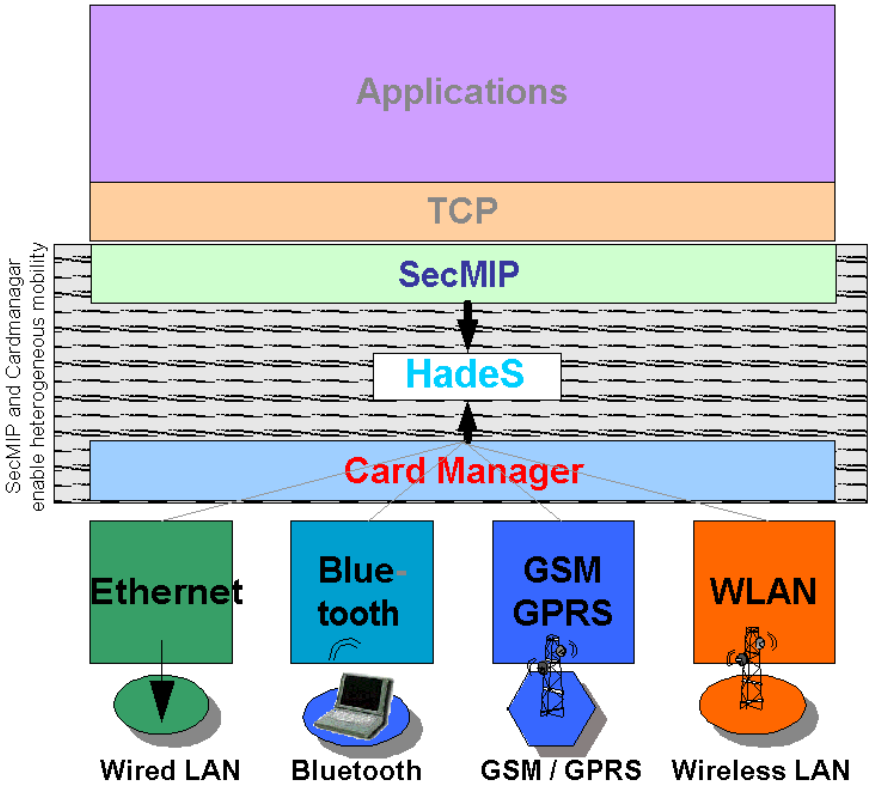


Figure 2.26 Adapted Network Layers

2.4.3 SECMIP MODULE

The SecMIP module consists of three interacting units. The MIP unit is handles Mobile IP procedures as Registration requests and replies. The IPSec module provides the mechanisms to establish an IPSec encrypted tunnel to protect the data exchanged between the mobile node and the home network (Mobile IP signalling messages and redirected data). The third unit is called SecMIP and controls the temporal operation sequence of Mobile IP and IPSec procedures.

To be independent of the underlying physical network device, the SecMIP module works with the virtual network device HadeS provided by the cardmanager (see Cardmanager Module). When this virtual device changes its IP configuration, SecMIP reacts by updating the IPSec tunnel endpoint and the mobile node's Mobile IP binding in the home network (home agent) according to the new IP configuration of the HadeS.

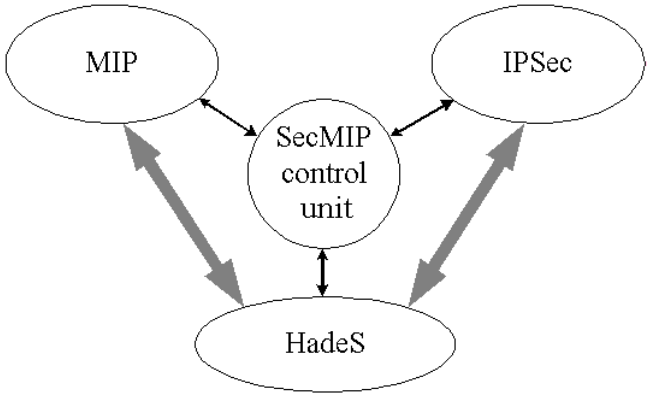


Figure 2.27 SecMIP

2.4.4 CARDMANAGER MODULE

The cardmanager module manages all available physical network devices of the mobile node. When a network device is physically added to the mobile node the cardmanager tries to lookup an IP configuration for this device. This configuration can be done either static by a configuration file or dynamically with the help of the DHCP service if available on the visited network. The cardmanager maintains a list of all well configured and working network interfaces. Based on different characteristics such as cost or bandwidth that the user can define, the cardmanager creates a rank list of all available devices. For the different available devices the user can define whether it can be used automatically or whether he has to be asked for. The device on the top of this ranking is linked with the virtual device HaDeS. When this ranking is changing and the link is updated to another physical interface, the SecMIP module reacts on this modification of the IP configuration as described above.

The cardmanager has to maintain the configurations of all devices. The pool of available devices has to keep up to date to ensure that the SecMIP module can always rely on a working HadeS.

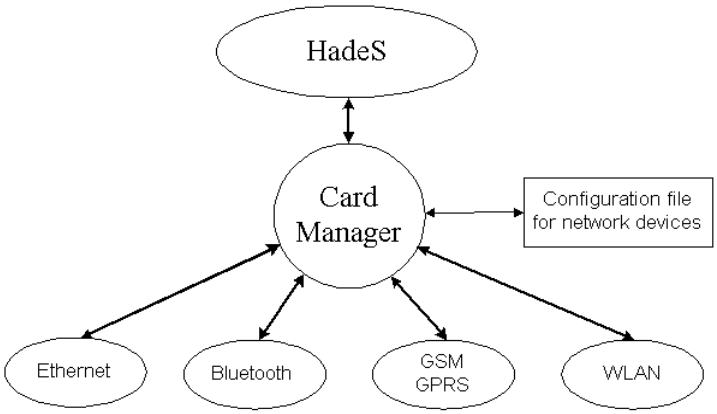


Figure 2.28 The Cardmanager

2.4.5 HARDWARE INDEPENDENT SECMIP DEVICE

The HadeS is the convergence point of the card manager and the SecMIP. As explained above, the idea is to make the physical change of device transparent for the IP layer. With the help of a virtual device that is linked to the chosen physical one, the SecMIP does not have to manipulate physical devices. And therefore, SecMIP keeps independent of new network technologies.

Compared with the normal Mobile IP proceeding, where a tunnel is build on the actual physical device, here the Mobile IP and the also the IPSec tunnel are build on the HadeS. This HadeS device takes the IP configuration of the underlying physical device.

2.4.6 WHERE DOES HADES FIT IN?

Figures 4 and 5 show how a normal Mobile IP routing is working. When the mobile node is at home, no tunnelling is deployed.

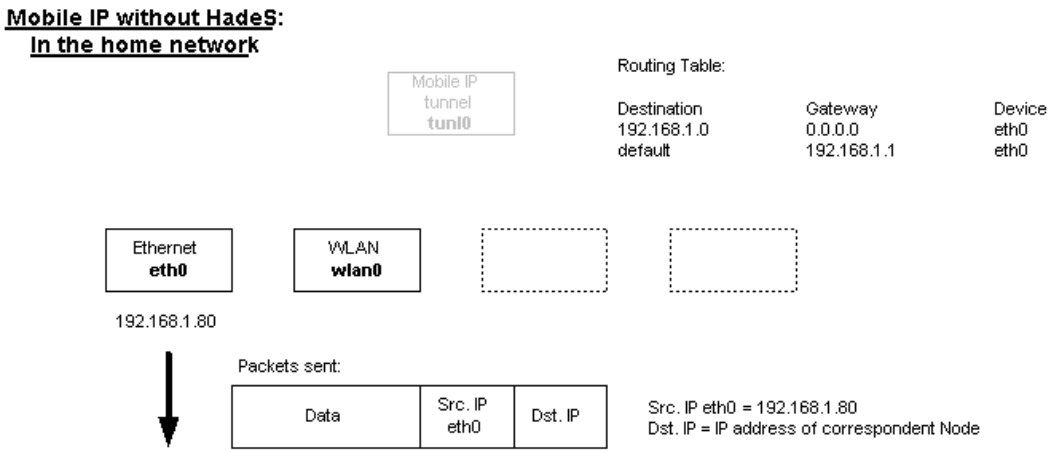


Figure 2.29 Mobile IP without HadeS in the home network

After being attached to a foreign network, a tunnel between the mobile node and its home agent is built. This tunnel device is built on the actual physical device (here the wireless device wlan0).

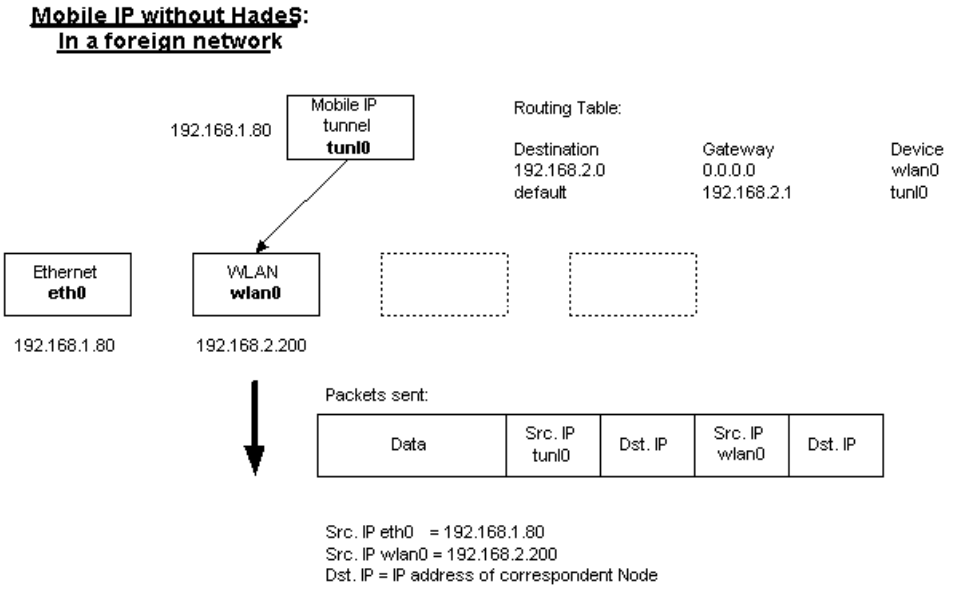


Figure 2.30 Mobile IP without HadeS in the foreign network

With HadeS as an intervening virtual device the routing looks different. Being at home, HadeS is configured with the home IP address. All communication passes through HadeS to the physical device.

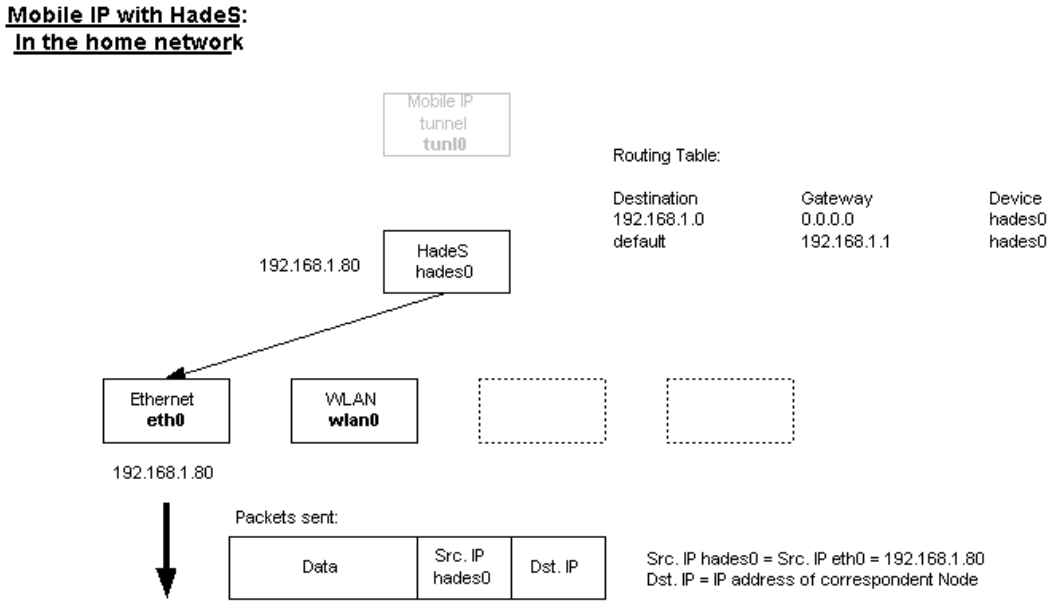


Figure 2.31 Mobile IP with HadeS in the home network

When changing access point, HadeS is linked to the new network interface and its IP configuration is updated according to the underlying device (here wlan0 with 192.168.2.200). The Mobile IP tunnel is now build on the virtual device HadeS to provide the home address to the applications.

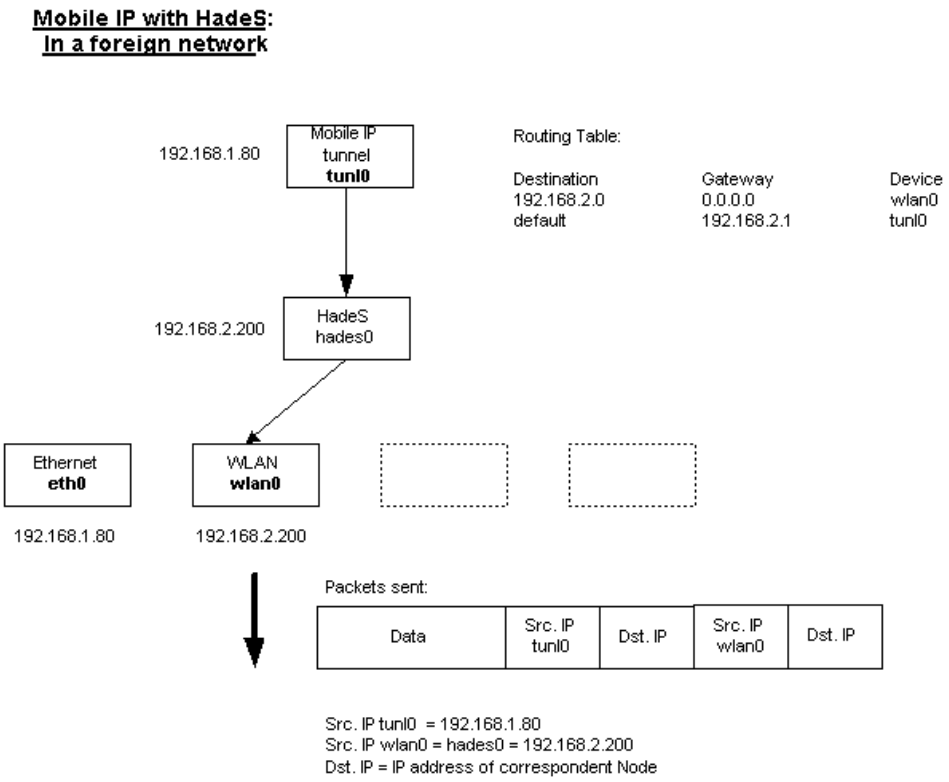


Figure 2.32 Mobile IP with HadeS in the foreign network

3

THE CARD CONFIG TOOL

3.1 REQUIREMENT SPECIFICATION

The main purpose of the Card Config Tool is to handle the different configurations of the network devices. The Card Config Tool is the part of the project whose responsibility will be to avoid handovers on mal-configured or un-configured devices. If an automated handover is performed and the device is not or not properly configured then no network connection will be available and the network related activities of an application will not be able to continue. For that reason the Card Config Tool is introduced. As a conclusion the following requirements for the Card Config Tool can be defined:

- The Card Config Tool must be present every time a PCMCIA related event is launched, e.g. when a card is inserted.

- The Card Config Tool must recognize already configured devices so that the user do not have to configure the same device twice and it must detect new devices which need to be configured and it has to alarm the user to add missing configurations.

- The Card Config Tool must offer an easily understandable user interface to give the user the possibility to add a missing configuration.

- The Card Config Tool must help to avoid mal-configurations of a device by performing some tests whether the configuration is valid or not.

- The Card Config Tool must be able to create system related files because the user should not have to change the system files himself to avoid errors and to improve the understandability of the product.

- The Card Config Tool must be able to store the different configurations of the known devices to a configuration database to avoid that the user must configure a device more than once.

- The Card Config Tool must offer the possibility to define different network scenarios to provide a bigger flexibility and a large usability of SecMIP.

- The Card Config Tool must have a Graphical User Interface to the configuration database so that the user can manage the different configurations and the different configuration scenarios.

- The Card Config Tool implementation must be flexible, so that new technologies and network can be easily integrated into the existing code. The more network technologies it supports the better the usability of SecMIP will be

3.2 DESIGN

The design process should lead to a precise plan how the product has to be implemented. With help of this plan, a proper and easy implementation must be possible. But this plan is not fix at one moment. Software development is a cycle, so that the design can change at implementing time. For that reason it is important to spend a lot of time for this phase in a project and to design carefully. The more stable a design is the less changes will occur and the less time will be lost for reengineering. For this reason this part of the documentation will be more voluminous than the section of the implementation.

First we have to decide which development technique will be chosen. To guarantee a flexible, scalable, maintainable and understandable implementation it is clear that this implementation will follow the aspects of an object oriented architecture.

The second decision that was taken is that we will focus on PCMCIA devices only. Most network devices on mobile computers are PCMCIA devices. This means not that there are no network devices apart these PCMCIA devices but in the first approach the focus will be on the PCMCIA ones. It would be a good task for future implementations of this project to support also non PCMCIA network devices and this aspect will surely be treated in the Outlook.

To understand how the Card Config Tool will work it is important to understand the PCMCIA process in GNU/Linux, cause we decided to focus on PCMCIA devices. The Card Config Tool has to be integrated into this process to provide its functionality. The PCMCIA related implementations in GNU/Linux were written and are maintained by David Hinds.

The following chapter has been taken from the PCMCIA-HOWTO of David Hinds and gives a good overview of the PCMCIA process in GNU/Linux. With the acknowledge about this process we can then define where the Card Config Tool must act and how it will act to satisfy the defined requirements.

The class diagrams and sequence diagrams of typical functions will complete this section of the documentation.

3.2.1 THE PCMCIA PROCESS IN LINUX

Each PCMCIA device has an associated "class" that describes how it should be configured and managed. Classes are associated with device drivers in */etc/pcmcia/config*. There are currently five IO device classes (network, SCSI, cdrom, fixed disk, and serial) and two memory device classes

(memory and FTL). For each class, there are two scripts in */etc/pcmcia*: a main configuration script (i.e., */etc/pcmcia/*scsi for SCSI devices), and an options script (i.e., */etc/pcmcia/scsi.opts*). The main script for a device will be invoked to configure that device when a card is inserted, and to shut down the device when the card is removed. For cards with several associated devices, the script will be invoked for each device. The *cardmgr* deamon which ships within the PCMCIA package has the responsibility to invoke these appropriate scripts. And to load the needed drivers for the inserted device.

The config scripts start by extracting some information about a device from the stab file. Each script constructs a "device address", that uniquely describes the device it has been asked to configure, in the *ADDRESS* shell variable. This is passed to the *.opts* script, which should return information about how a device at this address should be configured. For some devices, the device address is just the socket number. For others, it includes extra information that may be useful in deciding how to configure the device. For example, network devices pass their hardware ethernet address as part of the device address, so the *network.opts* script could use this to select from several different configurations.

The first part of all device addresses is the current PCMCIA "scheme". This parameter is used to support multiple sets of device configurations based on a single external user-specified variable. One use of schemes would be to have a "home" scheme, and a "work" scheme, which would include different sets of network configuration parameters. The current scheme is selected using the "cardctl scheme" command. The default if no scheme is set is "default".

As a general rule, when configuring Linux for a laptop, PCMCIA devices should only be configured from the PCMCIA device scripts. Do not try to configure a PCMCIA device the same way you would configure a permanently attached device. However, some Linux distributions provide PCMCIA packages that are hooked into those distributions' own device configuration tools. In that case, some of the following sections may not apply; ideally, this will be documented by the distribution maintainers.

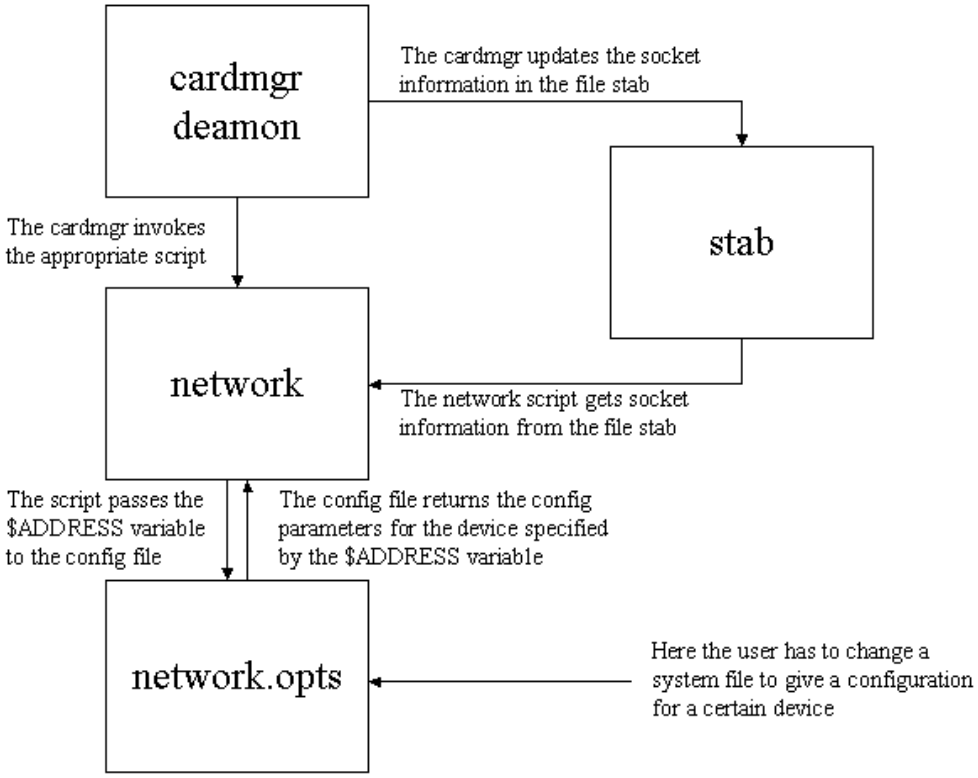


Figure 3.1 The PCMCIA process in Linux

This implementation of the PCMCIA behavior in GNU/Linux causes two important problems for an automated handover. The first problem is that the user must change a text file to configure the network devices. This is never a good way if a user friendly and widely automated product is developed. Often such files are not documented and without an effort to read the documentation of the whole software package or a specific HOWTO these files are heavily understandable. This causes errors. These text files often do not provide any test routines for the data given by the user and after all it is not user friendly when a text file must be edited, in order to get an application working properly. The Card Config Tool offers a solution to this problems by providing a wizard which makes it possible even for inexperienced users to configure their network devices.

A second problem is the fact, that the PCMCIA process do not warn the user if a device setup fails. It is only logged and if the user is not experienced enough he will not be able to detect why a handover on this devices will fail. The PCMCIA only invokes *ifconfig* to bring the device up. The Card Config Tool in contrary pauses the PCMCIA process, checks the database for an appropriate configuration, which it will do immediately with the user, when not available, and resumes the PCMCIA process.

By performing these two functionalities the Card Config Tool can avoid that the whole application tries to do a handover on a mal-configured or un-configured device by simply shunning such bad configurations for a certain device. For that reason the Card Config Tool has to be integrated in the PCMCIA process to act on the root of these two problems and to solve them. The following chapter will explain how and where the Card Config Tool is acting.

3.2.2 CARD CONFIG TOOL AND PCMCIA PROCESS INTERACTION

The Card Config Tool interacts in this process in order to avoid bad configurations or to avoid that a device is not configured, when a handover is performed.

The *cardmgr* deamon , a build in component of David Hinds PCMCIA package, catches events that are launched when a socket reports a change. In our case this means when a device is added or removed. This cardmgr detects what kind of PCMCIA device is treated and launches the appropriate *script* which will get a configuration, if one is available, for that device and will bring up the device if a configuration is available and if this configuration is valid.

The problems mentioned in the previous chapter reside between the user and the *.opts* file and between the *script* and the *.opts* file. So the Card Config Tool has to be introduced there in order to perform its tasks.

The Card Config Tool acts between the *script* (e.g */etc/pcmcia/network* if it's a network device) and the appropriate *.opts* file (in this example *network.opts*). It is invoked by the *script* the *cardmgr* launches when a card is inserted or removed. The Card Config Tool pauses the PCMCIA process, tests the availability of a configuration for this device in a database and if no configuration is available the Card Config Tool raises a wizard which will guide the user through the necessary steps to build such a configuration. So for new devices no manual text file changing will be necessary any more. This text files will be generated by the Card Config Tool. Then the PCMCIA process continues and will lead to a properly working device. The Card Config Tool must so be invoked by the *script* and that for these *scripts* need to be changed in the following way:

```
network:


...


start_fn () { return; }
stop_fn () { return; }

case "$ACTION" in
```

```
'start')

    xterm -display 0:0 +hold -e python /pathto/cardtool/test.py $ADDRESS
$DEVICE
    ;;

esac
. $0.opts
...
```

serial:

```
...
start_fn () { return; }
stop_fn () { return; }

case "$ACTION" in

'start')
    xterm -display 0:0 +hold -e python /pathto/cardtool/test.py $ADDRESS
$DEVICE
    ;;

esac
. $0.opts
...
```

The red lines must be inserted in these *scripts* to launch the Python script *test.py* which will take control to do the defined tasks.

The first line means, that if a card is inserted and only then the Card Config Tool Control Unit (*test.py*) will be invoked. The Control Unit must extract some information out of the *ADDRESS* shell variable and so this variable is passed as well as the device name (e.g. eth0) to the Card Config Tool, which then will pause the PCMCIA process and perform the above explained tests before it continues the PCMCIA process.

The following picture illustrates how and where the Card Config Tool acts:

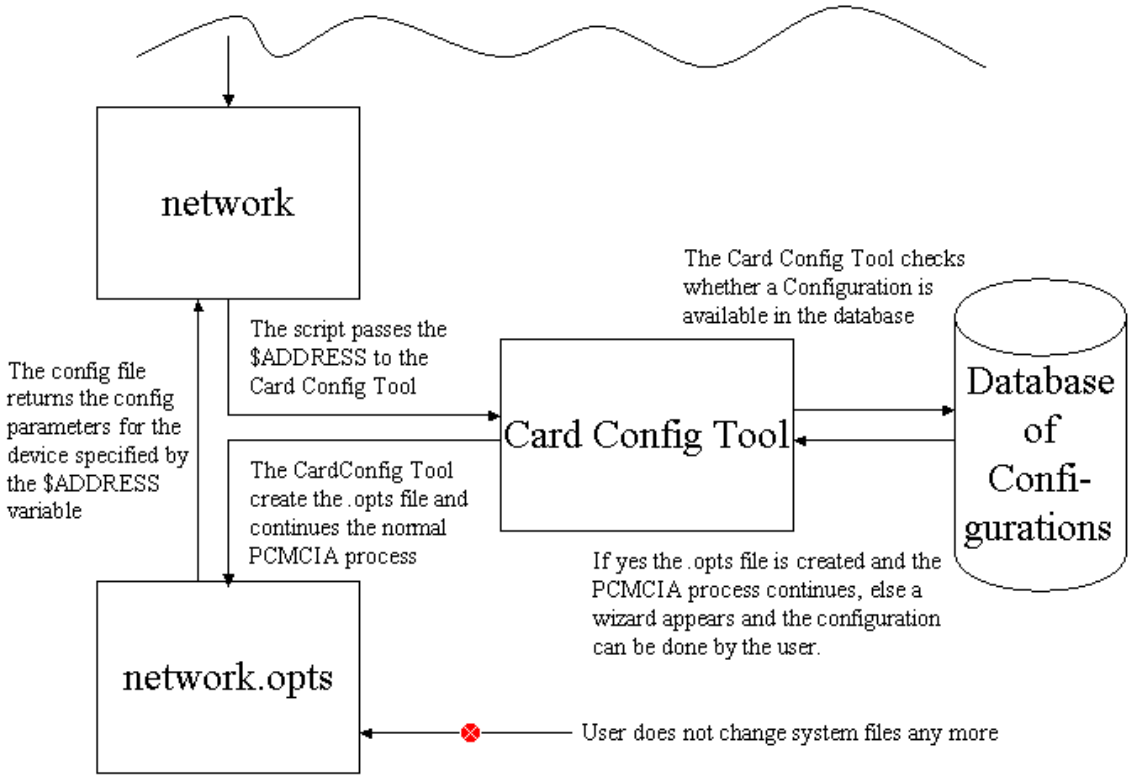


Figure 3.2 How the Card Config Tool interacts with the PCMCIA process in GNU/Linux

3.2.3 THE BASIC STRUCTURE OF THE CARD CONFIG TOOL

The functionality of the Card Config Tool can be divided in 2 logical parts. The first part is the part that is responsible to check if a running PCMCIA process will lead to a successful setup of a network device. This part of the Card Config Tool is called Control Unit and is invoked indirectly by the cardmgr through the *scripts*. It is important that this part stops the PCMCIA process, tests the available configurations and if none is available it has to invoke the wizard to help the user to make a configuration. Then it must resume the PCMCIA process which will result now in a successful setup of the device.

The second part, the Config Manager, is the part that offers the user the possibility to manage existing configurations or to add new configurations for certain devices. In this part it is also possible to define different configuration scenarios for given devices. This could be useful to apply a different configuration on the same device e.g when we change from one wireless network to another, it will then be enough to change the scenario. It will be the responsibility of the JorDI section of the project to force such scenario changes. This second part can be launched by the user and is that for independent of the first part of the Card Config Tool.

These two sections of the Card Config Tool share a third component. This component has nothing to do with the logical cycle of the tool, a wizard. It is a hint for the user to configure a device properly. This wizard shows up every time when the first part recognizes that there is no existing configuration for the treated device available. E.g if a new network device is inserted, for which the Card Config Tool has no configuration in the database, this wizard will appear and the user can configure the device. The configuration is now stored in the database and the device will be brought up correctly. In this wizard simple tests are integrated to test whether the information the user passes to the Card Config Tool is valid.

For the second part the wizard can be used to change an existing or to add a new configuration for a device. If the user adds a new configuration then the wizard will have the same appearance as it has, when it is invoked when a configuration is needed in the PCMCIA process. If the user wants to edit an existing configuration, then the wizard will appear with the values that are stored in the database for that configuration, so that the user do not have to give the whole information again, if he just wants to change an IP address.

The picture below should just give an idea on how the responsibilities of the Card Config Tool are distributed to provide the needed functionality.

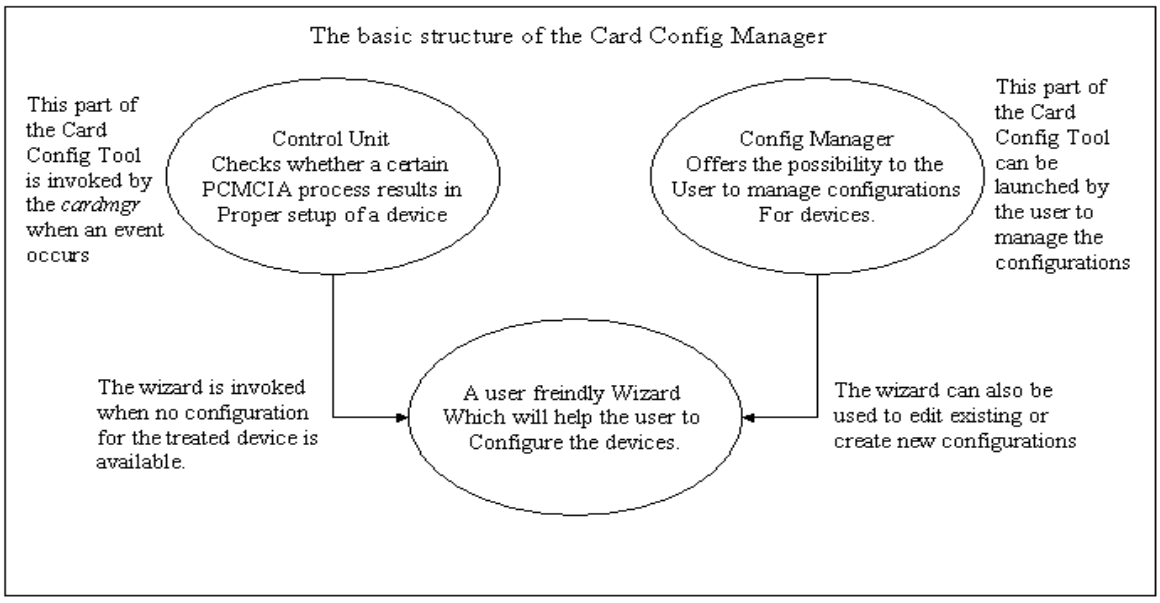
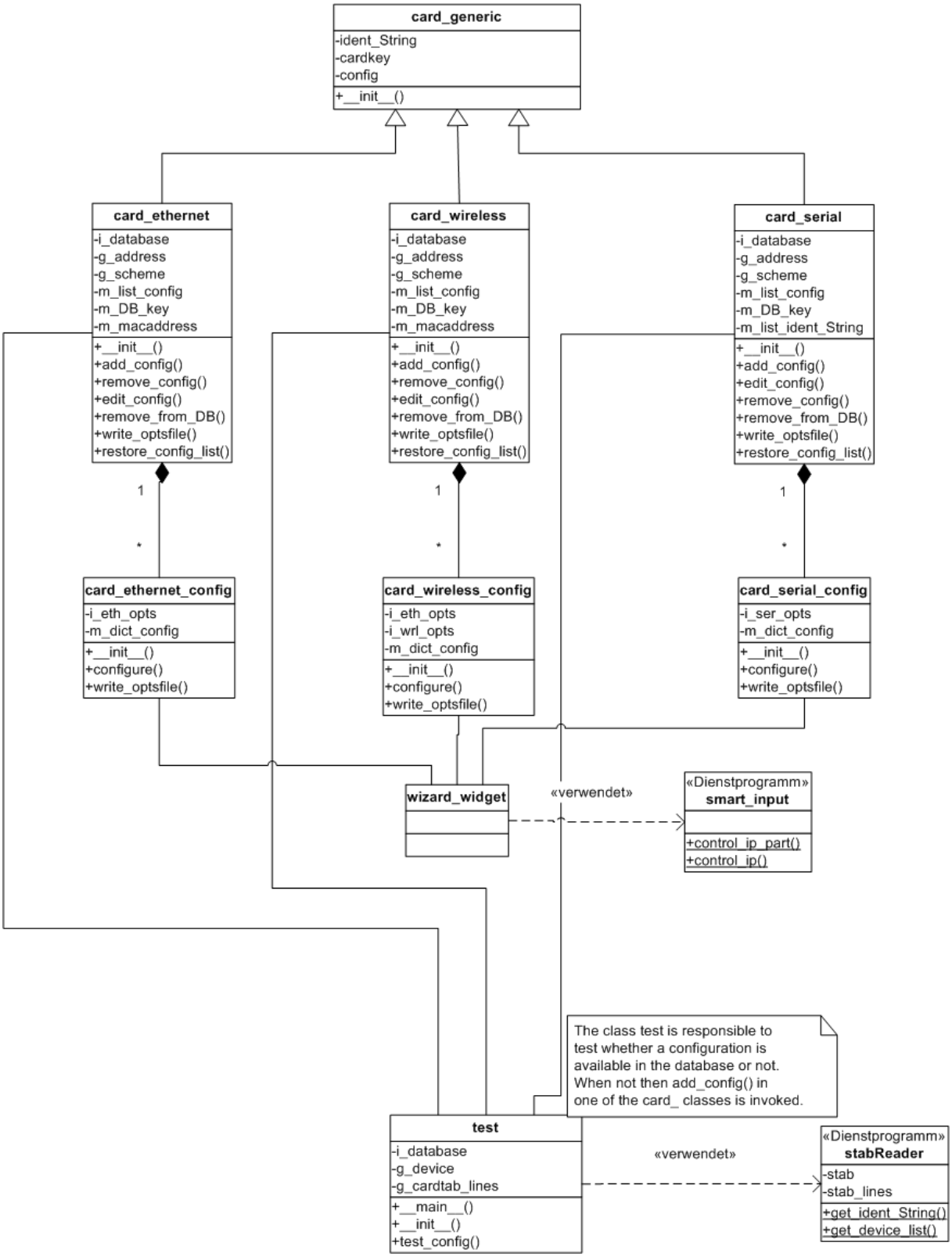


Figure 3.3 The basic structure of the Card Config Tool

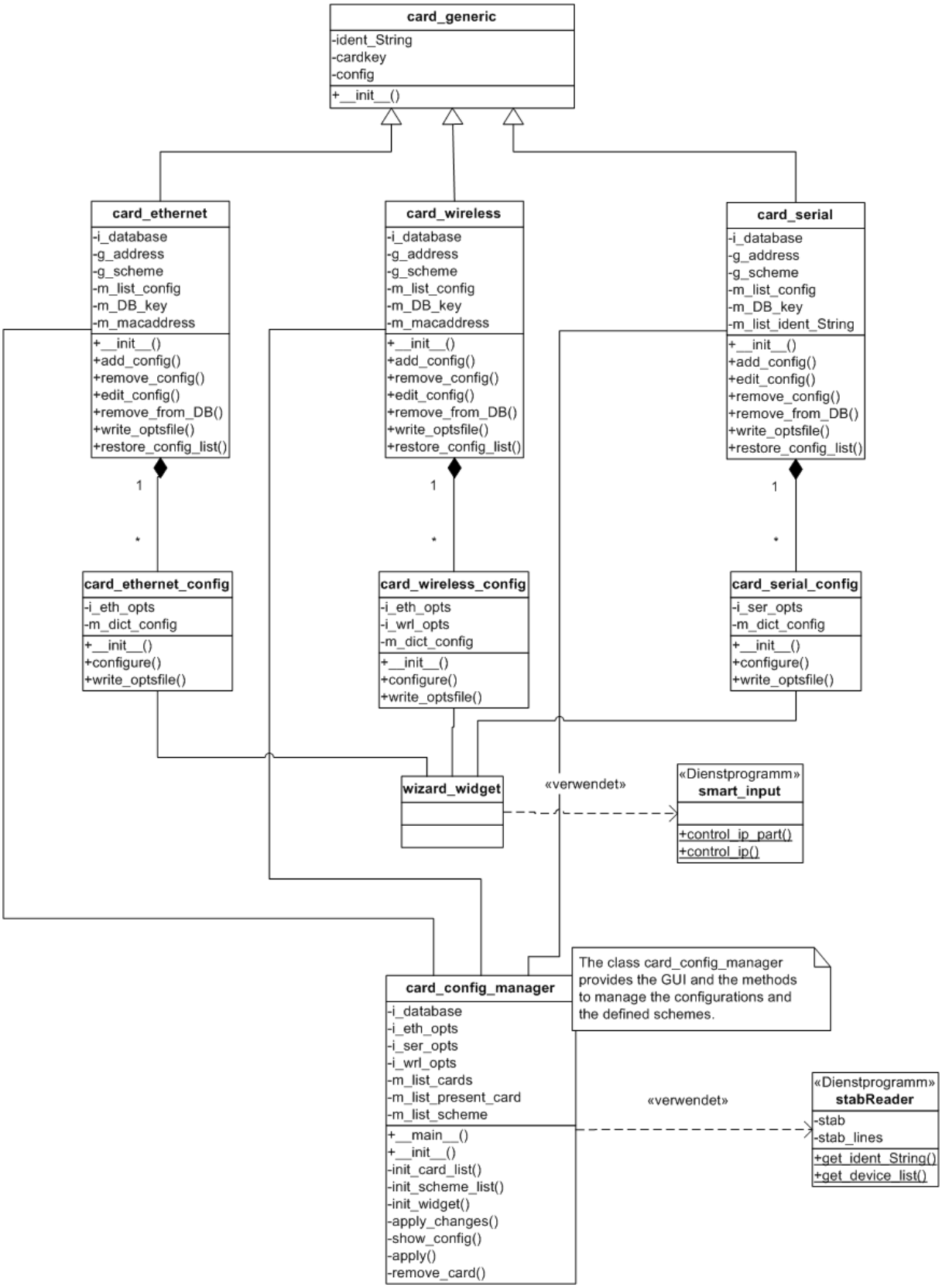
3.2.4 CLASS STRUCTURE OF THE CARD CONFIG TOOL

The Control Unit



Class Diagram 3.1 The Control Unit

The Config Manager



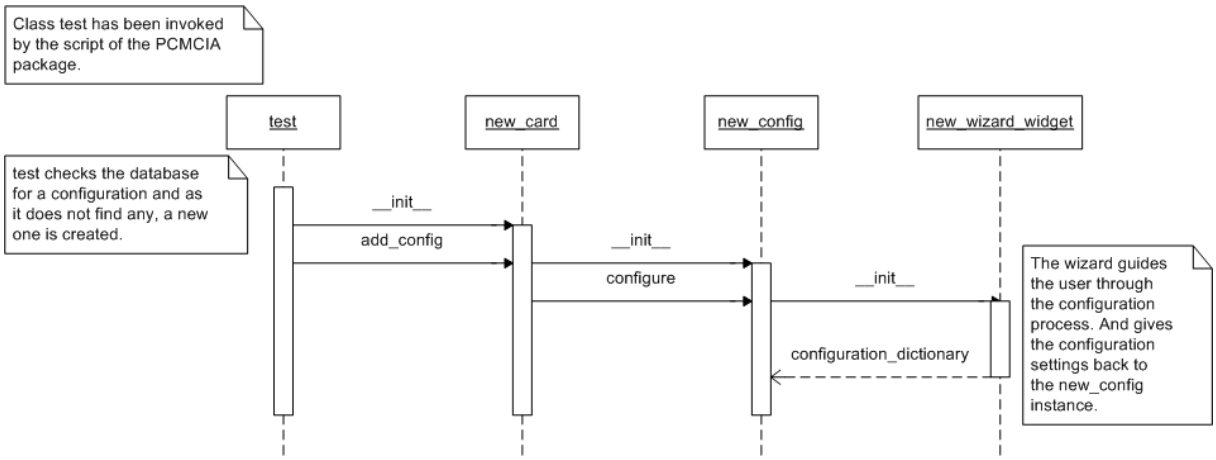
Class Diagram 3.2 The Config Manager

3.2.5 EXPLANATIONS OF THE CHOSEN DESIGN

After this dry diagrams it is time for some explanations. The decision to divide the class diagrams in two separate diagrams was made to show how the two parts of the Card Config Tool, the Control Unit and the Config Manager work. It is clear that this two parts are integrated in one product and that the division is only for explaining purpose. The classes *card_generic*, *card_Ethernet*, *card_wireless*, *card_serial*, *card_ethernet_config*, *card_wireless_config*, *card_serial_config*, *wizard_widget*, *smart_input* and *stabReader* are the same classes in both parts. Two diagrams only to show how the class *test* and the class *card_config_manager* interact with them.

Explanation of all functions would go beyond the scope of this document and that for some sequence diagrams should show the use of them.

Note: the functions and attributes of the wizard are not documented because it is useless for the explanation of the logic of the program.



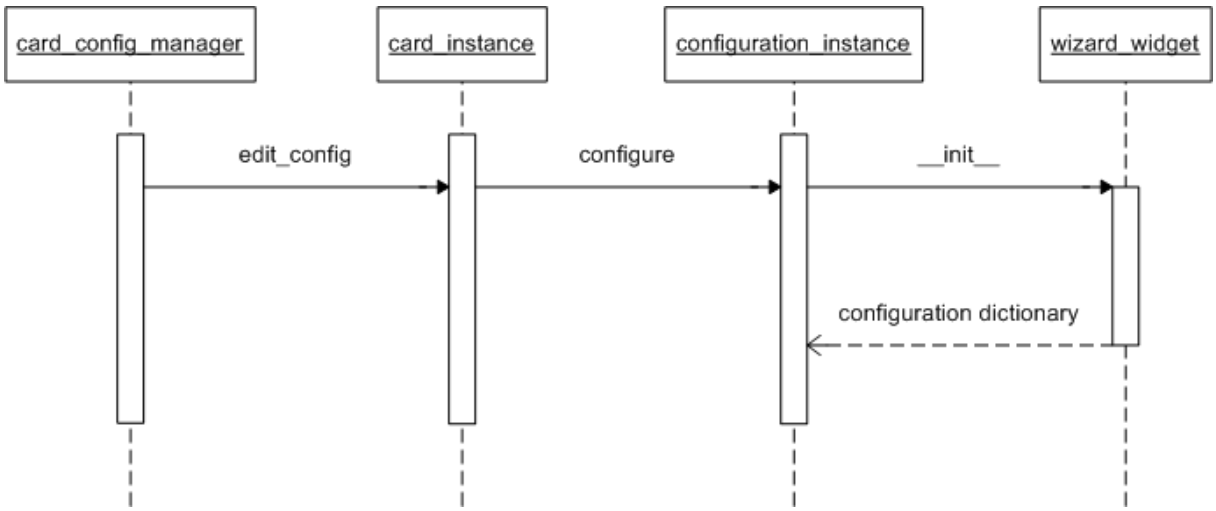
Sequence Diagram 3.1 Negative test and following configuration of a card

In this sequence diagram we can see the process of adding a new card and adding a new configuration for that card. This happens, when the Card_config Tool detects that a card is entered that is not in the database. So the card is unknown for the Card Config Tool. The Control Unit now create a new card instance, for of the kind of card that can be determined out of the *ADDRESS* shell variable (e.g. Ethernet). After having created that instance The Card Config Tool initiate the configuration for that card instance.

This means that this new card instance creates a new configuration intstance. This instance creates a new wizard instance, which provides the GUI to do the configuration. A configuration dictionary is created and his values are passed to the configuration instance which will store it. Now the

configuration is append to the list of configurations of the card instance. In the case it is a new card this will be the first configuration for that card.

Then the card with his configuration is stored in the database. The Control Unit then creates the *.opts* file and ends. The PCMCIA process continues and the new configuration will be applied.



Sequence Diagram 3.2 Editing an existing configuration with the Configuration Manager

This sequence diagram shows the process when a user edits a configuration for a certain card using the Config Manager. In the Config Manager the user has to chose the card. A list of available configurations appears. Every configuration has his own scheme. Then the user can chose the configuration he wants to change and he clicks edit.

The Configuration Manager calls the function *edit_config(configID)* of the card instance the user has chosen. As parameter of this function the Configuration Manager gives the *configID*. The card instance now calls the function *configure* of the configuration instance with the given *configID*. In this function a new instance of the wizard will be created. This wizard will appear with the settings that are currently made for this configuration instance.

Note that one can not change the scheme of a configuration. If the scheme must be changed then the configuration with the scheme, that has to be changed must be deleted and a new one must be created with another scheme name but the same settings. This is because the scheme is the primary key for the database. For the reason to keep this database consistent it is dangerous to change the key.

With these two sequence diagrams it should be possible to understand the design.

3.3 IMPLEMENTATION

In this chapter the implementation of the first approach of this project is explained. This implementation is not a mature product. It is a tool developed for testing purposes and prototyping. That for there are a lot of lacks in this implementation and many aspects will not be treated. This implementation does not claim to be powerful, fast or a complete implementation of the design defined in the last chapter. So this implementation will differ from the theoretical design decisions. There a lot of assumptions were made to facilitate the realization and to avoid missing the goal by going to deep into a certain subject. It was the aim to show that the design can be implemented and that it satisfies the defined requirement specification. After all the Card Config Tool is working fine and is used in the labor to configure devices because its easier than changing text files. It also avoids that the user forgets to configure a device what was often useful for the developers. This means that the implementation has reached the aims and it is still close enough to the design to prove that this design is a good approach.

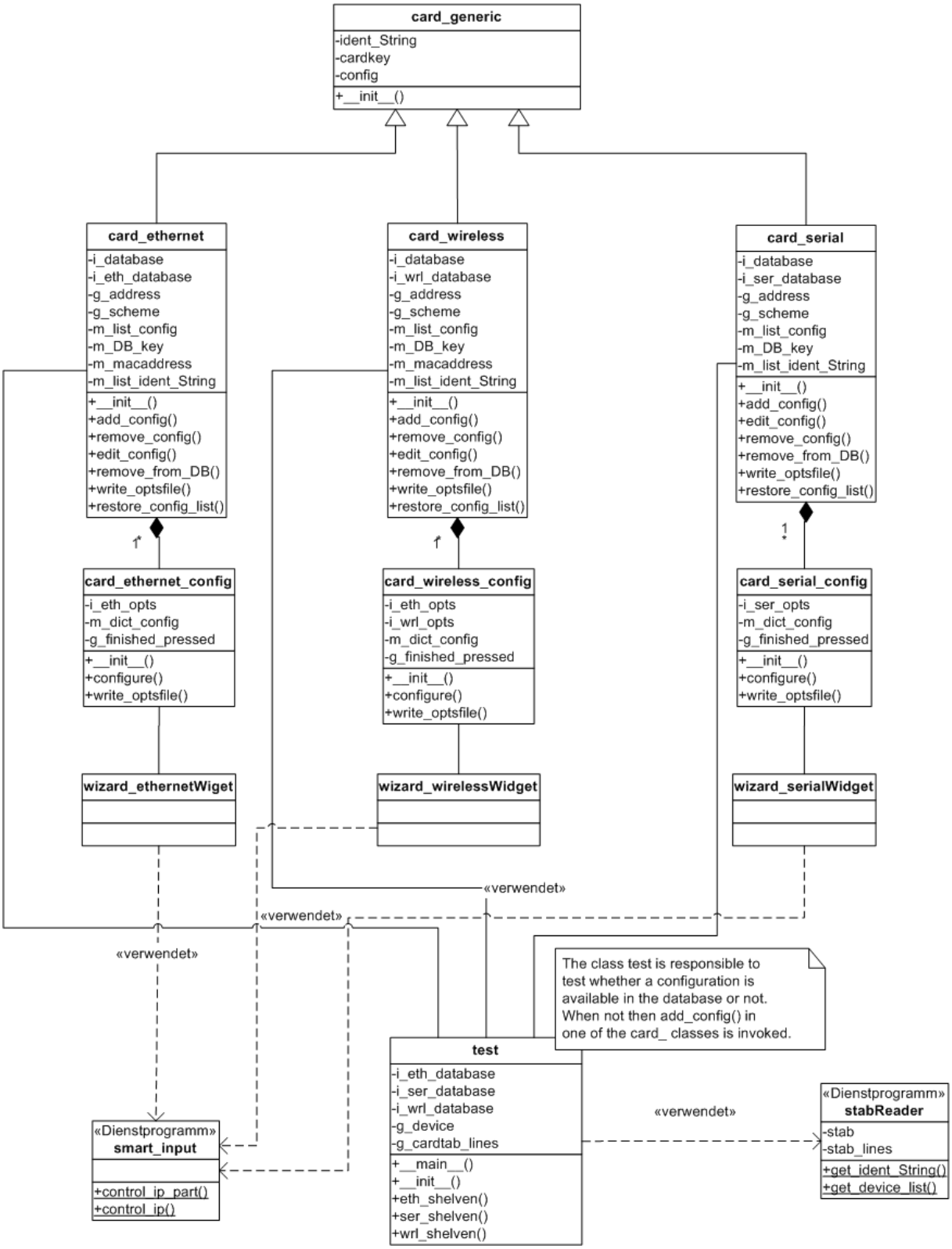
To visualize these aberrances from the design the class diagrams of this implementation will be showed and explained where the differences are and why they are there. Most likely they are small but in order to have a detailed documentation they are explained. The biggest change was a conclusion of the decision not to use an SQL database and to use Python as programming language. Python provides with his pickle class a fast and simple way to store objects in normal files. This avoids to have a running SQL server on a mobile computer which is probably not configured for that. To store the card instances with all their configuration objects it was important to share the data on different files instead of storing them in tables. But a SQL database realization can still easily be implemented.

In the second part the development environment will be explained. This means in particular the chosen programming language, the used toolkit and the utilities that have been taken to make certain tasks easier to develop, like building a graphical user interface. This is done by giving some basic information on what it is and by telling why a particular choice has been made.

For developers there are a lot of "little helpers" available and the wide spectrum of these tools made it impossible to test different tools with the same function against each other. This means that in this document no solutions about chosing a certain tool to provide certain functionaties can be found.

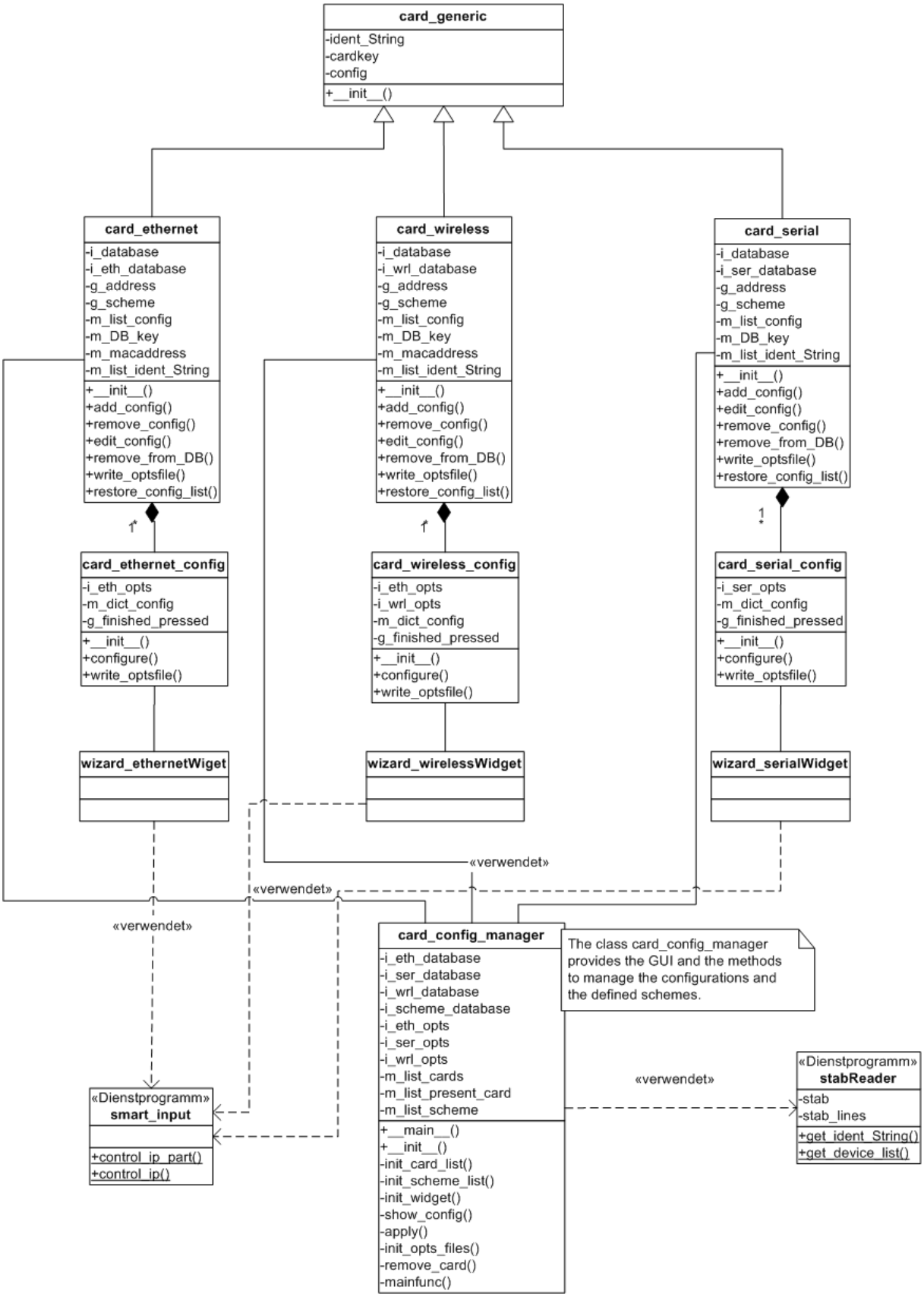
3.3.1 CLASS DIAGRAMS OF THIS IMPLEMENTATION

The Control Unit



Class Diagram 3.3 The Contorl Unit

The Config Manager



Class Diagram 3.4 The Config Manager

3.3.2 ABERRATIONS FROM THE DESIGN

In the present implementation of the Card Config Tool some aberrations of the original design are realized in order to get a fast implementation for testing purposes.

First the GUI must be mentioned. The decision was made to create a wizard for every kind of network device. The priority was not focusing on the graphical user interface. Important was that the logic of the program is working. The GUI is only important for visualization issues. In the requirement specifications is the GUI defined as an important part of the product. So a GUI is provided but not mature. In a future release this GUI has to be reengineered. A second reason was that with the use of Glade is was not possible to implement the design. But the use of Glade was important in order to get fast results and for demonstrating purposes. The part of the GUI is not a good code, but it is totally isolated from the rest of the implementation so that it is easily possible to take the existing version of the GUI out of this implementation and to replace it by a good and well implemented one.

The second aspect that has to be treated is the fact that no SQL database has been used. It is not defined in the requirement specification to use such a database but for performance increase it would be better to use it. The reasons not to use a SQL database were first not to force the client to have an SQL server running, and then that Python with his pickle class offers a good alternative to store objects. So this pickling mechanism is used to store the cards and their configurations. For this reason in the class diagram of the implementation show more attributes up than in the ones of the design. These attributes contain the paths to the different files where Python pickles the objects to. More than one file are necessary to store all the objects.

3.4 DEVELOPING TECHNOLOGIES USED

The first approach of the Card Config Tool is entirely implemented in Python. This is an object oriented language with a lot of scripting capabilities and support for the GTK (Gimp ToolKit).

The GTK provides the widgets we use to build the graphical user interfaces for the Card Config Tool. The GTK has been chosen because besides his comfortable look and feel it is supported by the most window managers which are available for GNU/Linux.

To build the graphical user interface of the wizard and of the configuration manager Glade has been used. Glade is a tool where you can assemble GUIs with mouse clicks. Glade generates then a XML code file which can be translated by a little utility named Glade Python Code Generator to Python code.

3.4.1 GNU/LINUX

Linux is an operating system that was initially created as a hobby by a young student, Linus Torvalds, at the University of Helsinki in Finland. Linus had an interest in Minix, a small UNIX system, and decided to develop a system that exceeded the Minix standards. He began his work in 1991 when he released version 0.02 and worked steadily until 1994 when version 1.0 of the Linux Kernel was released. The current full-featured version is 2.4 (released January 2001) and development continues.

Linux is developed under the GNU General Public License and its source code is freely available to everyone. This however, doesn't mean that Linux and its assorted distributions are free -- companies and developers may charge money for it as long as the source code remains available. Linux may be used for a wide variety of purposes including networking, software development, and as an end-user platform. Linux is often considered an excellent, low-cost alternative to other more expensive operating systems.

Due to the very nature of Linux's functionality and availability, it has become quite popular worldwide and a vast number of software programmers have taken Linux's source code and adapted it to meet their individual needs. At this time, there are dozens of ongoing projects for porting Linux to various hardware configurations and purposes.

S.u.S.E Linux is one of these Linux distributions mentioned above. The actual release and the release used in this project is S.u.S.E Linux 7.2 Professional. S.u.S.E Linux 7.2 Professional, the

well-known collection of more than 2000 up-to-date Linux tools and software packages, is designed for professional settings and for technically advanced users.

With S.u.S.E Linux, both home users and technically advanced users can easily implement network and security solutions as well as data exchange in heterogeneous networks. Graphical user interfaces such as KDE and GNOME, the Logical Volume Manager, the Reiser File System and the ADSL support meet all requirements of complex IT environments. Database solutions, server applications, and development tools complete the S.u.S.E Linux 7.2 Professional software package. Even Internet-based video and voice telephony or groupware solutions can be implemented with S.u.S.E Linux 7.2 Professional.

A special treat for home users: popular desktop applications like StarOffice 5.2, Netscape Navigator, the innovative universal browser Konqueror, GIMP, Moonlight 3D, and Broadcast2000 turn your PC into a high-speed multimedia workstation. A video editing program and professional sound and MIDI tools are also included.

Sources: http://www.linux.org & http://www.suse.com [7], [8]

The decision to use GNU/Linux to develop our product is easy to explain. We need a free and open environment, cause we have to go deep into the kernel of the OS. The availability of the source code makes it possible to change certain aspects of the OS for our needs like the system scripts mentioned in the design part of this document.

A second reason was that the implementations of Marc Danzeisens SecMIP is running on a GNU/Linux system.

As distribution S.u.S.E Linux was chosen because it ships with the original PCMCIA package of David Hinds and because most computers in our laboratory are configured with S.u.S.E Linux. The distribution was available and so no extra money for other distributions had to be spent. Other distributions handle the PCMCIA process differently and it could be a future task to get the Card Config Tool working on them.

3.4.2 PYTHON

Python is a general purpose programming language. Python may also serve as a glue language connecting many separate software components in a simple and flexible manner, or as a steering language where high-level Python control modules guide low-level operations implemented by

subroutine libraries effected in other languages. Python is also easy to learn and use, so it could also serve as an interface for naive users of advanced applications, or as a first programming language.

Some qualitative properties of Python:

- Python is easy: Python's simple syntax and elegant and clean semantics is easy for programmers and non-programmers to learn, read, and use. Experienced Programmers will be productive in Python in a day. Novices might be productive in Python in a day and a half. Python's basic syntax looks like a radical simplification of the algol/C/Pascal programming languages and as such is easily learned by people who have some experience with common mathematical notations and/or other common programming languages.

- Python is extensible and flexible: Python can be extended easily to interface with other software systems, and Python can also be easily incorporated into other programs as a component. Furthermore, Python allows extreme flexibility in the treatment of language components.

- Python is interpreted: Python is dynamically interpreted language that supports byte compilation. Python programs may be developed and tested with the help of the interactive mode of the Python interpreter which allows program components to be debugged, traced, profiled, and tested interactively. Python byte code is also machine independent and may be executed on different hardware and software platforms without recompilation.

- Python is object oriented: Python supports object-oriented class structures with multiple inheritance and late binding. Python is also truly object oriented to the core: Everything in Python is an object organized in a flexible and general internal object framework that is beautiful to behold. Experienced C programmers who delve into the source code of the Python distribution will find the core implementation a delightful read. Should the need arise, programmers can create alternate implementations for any basic component of the Python language (such as numbers or even object classes) either in Python code implementations or via compiled language extensions.

- Python is stable, tested, and upwardly compatible: Python always has been upwardly compatible through the years and will continue to be upwardly compatible. New versions of the interpreter will always run programs written for old versions of the interpreter. At this point new versions of the interpreter include precious few bug fixes, because the

Python core has been thoroughly tested and debugged in thousands of applications for the last several years. Most bugs found these days are obscure and avoidable.

- Python is freely available with unrestricted redistribution in source form: The Python copyright essentially protects the authors from legal jeopardy and prevents malicious users from attempting to hijack the copyright. Aside from that, Python programmers and users may use Python in source or binary form just about anyway they please. In particular, programmers may create products that use Python and release the product in binary-only form with all Python modules in byte-compiled-only form, and they may sell or give away the result in any manner they think will make them the wealthiest, or the most famous.

Sources: http://www.fsbassociates.com/books/pythonchpt1.htm [9]

Python as programming language was chosen because of its scripting capabilities. This made Python perfect to use with GNU/Linux. Python was preferred to Perl because Python is easier to learn. Python supports all object-oriented features to realize the given design. After all Python provides with his pickle class a good alternative to a SQL database.

3.4.3 GTK

GTK (GIMP Toolkit) is a library for creating graphical user interfaces. It is licensed using the LGPL license, so you can develop open software, free software, or even commercial non-free software using GTK without having to spend anything for licenses or royalties.

It's called the GIMP toolkit because it was originally written for developing the GNU Image Manipulation Program (GIMP), but GTK has now been used in a large number of software projects, including the GNU Network Object Model Environment (GNOME) project.

GTK+ has a C-based object-oriented architecture that allows for maximum flexibility, and consists of the following component libraries:

- GLib - Provides many useful data types, macros, type conversions, string utilities and a lexical scanner.

- GDK - A wrapper for low-level windowing functions.

- GTK - An advanced widget set.

GTK is built on top of GDK (GIMP Drawing Kit) which is basically a wrapper around the low-level functions for accessing the underlying windowing functions (Xlib in the case of the X windows system).

GTK is essentially an object oriented application programmers interface (API). Although written completely in C, it is implemented using the idea of classes and callback functions (pointers to functions). There are GTK bindings for many other languages including C++, Guile, Perl, Python, TOM, Ada95, Objective C, Free Pascal, and Eiffel.

Sources: http://www.gtk.org/ & http://www.gtk.org/tutorial/ch-introduction.html [10], [11]

GTK was used because of his nice look-and-feel and the GTK is a mature product. The GTK is supported by the most window managers for the Xfree86 Server running on most GNU/Linux systems. And after all there exists a binding for Python.

3.4.4 GLADE

On the home page of Glade the following sentence can be read: "Glade is a free user interface builder for GTK and GNOME. It is released under the GNU General Public License (GPL)." It is useless to explain more about it. There for the link of its home where a lot of nice screenshots can be seen:

http://glade.gnome.org/ [12]

Glade has been chosen because it is a very time consuming process to develop a GUI without using it.

3.4.5 GLADE-PYTHON CODE GENERATOR

Python is proving to be a popular scripting language for implementing system functions for the linux operating system. GTK is the toolkit used by the Gimp and a number of other programs including GNOME. Glade is a GUI builder that allows for rapid GUI prototyping for GTK. James Henstridge has produced python bindings for GTK as well as a runtime library for glade xml files. The python code generator is a compile time approach to the same end. It parses a glade xml file and produces runnable python code that uses the python bindings for GTK. This code is freely distributed under the GPL.

Source: http://glc.sourceforge.net/ [13]

This code generator has been chosen to build automatically Python code out of Glade. This safes time.

3.5 LAST BUT NOT LEAST: HOW DOES IT LOOK LIKE?

We can only show the Config Manager and the Wizard. The Test Unit does not have a GUI.

First the Config Manager will be shown. It is the main window with the list of the configured cards. Here the user has the possibility to change or add new configuration of recognized cards. In future releases he will also be able to chose the configuration scheme and the handover decision.

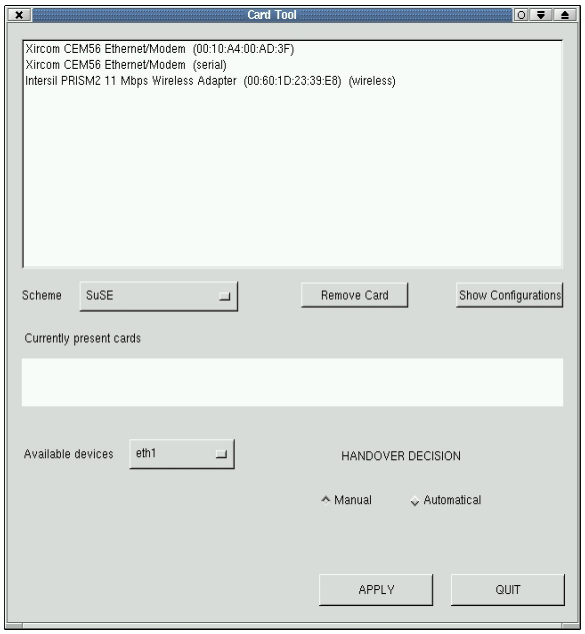


Figure 3.4 The Config Manager

In this window the user selects the device and can delete it from the internal list (note: if the user deletes a device the Control Unit will re-recognize it the next time it is inserted and claim a configuration) or have a look at the different schemes for which a configuration exists. In this example the user decided to look at the schemes:

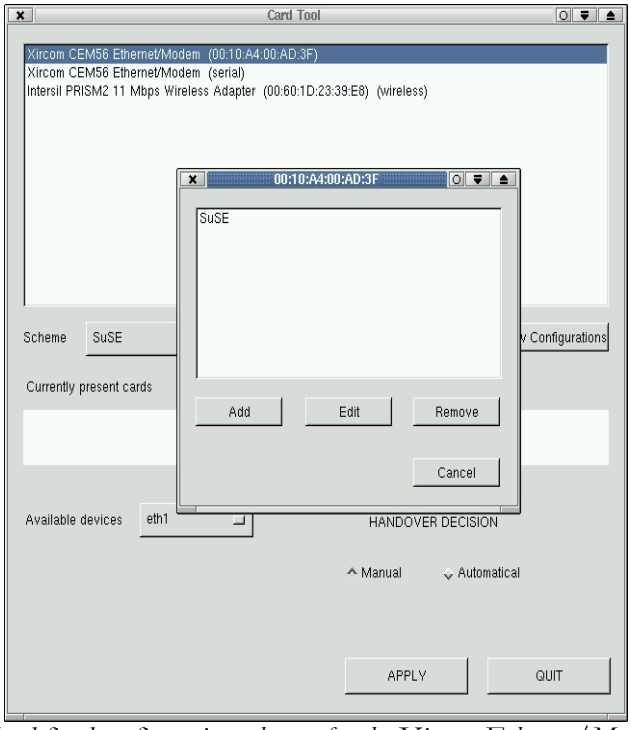


Figure 3.5 The defined configuration schemes for the Xircom Ethernet/Modem Adapter

And the last thing shown here is the Wizard which helps the user to do a configuration:

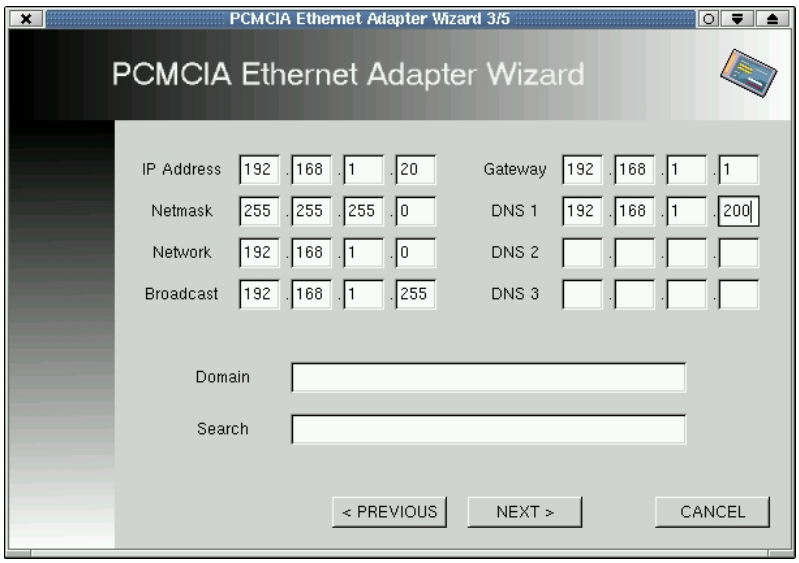


Figure 3.6 The IP configuration page of the Wizard.

This is the third page of the Wizard for an Ethernet device. Here the user can specify the IP configuration for his device. The Wizard offers some basic tests of these settings, too so that the process will lead to a successful configuration and can be brought up correctly by the GNU/Linux PCMCIA implementation.

4 HADES

4.1 HADES: HARDWARE INDEPENDENT SECMIP DEVICE

In order to protect the product from possible competitors a patent for a method to implement the concept of full IP Mobility was submitted. It was called HadeS (the God of the Underground) and in this chapter we define the concept and the real implementation under Linux. First of all the mechanism is described, then some concepts related to kernel programming are presented to finally end up in the details of the implementation.

4.2 THE CONCEPT

HadeS is the convergence point of the Card Config Tool and the SecMIP. As explained above, the idea is to make the physical change of device transparent for the IP layer. With the help of a virtual device that is linked to the chosen physical one, the SecMIP does not have to manipulate physical devices. And therefore, SecMIP keeps independent of new network technologies.

Compared with the normal Mobile IP proceeding, where a tunnel is build on the actual physical device, here the Mobile IP and also the IPSec tunnel are built on the HadeS. This HadeS device takes the IP configuration of the underlying physical device.

4.3 WHERE HADES FITS IN

Figures 4.1 and 4.2 show how a normal Mobile IP routing is working. When the mobile node is at home, no tunnelling is deployed.

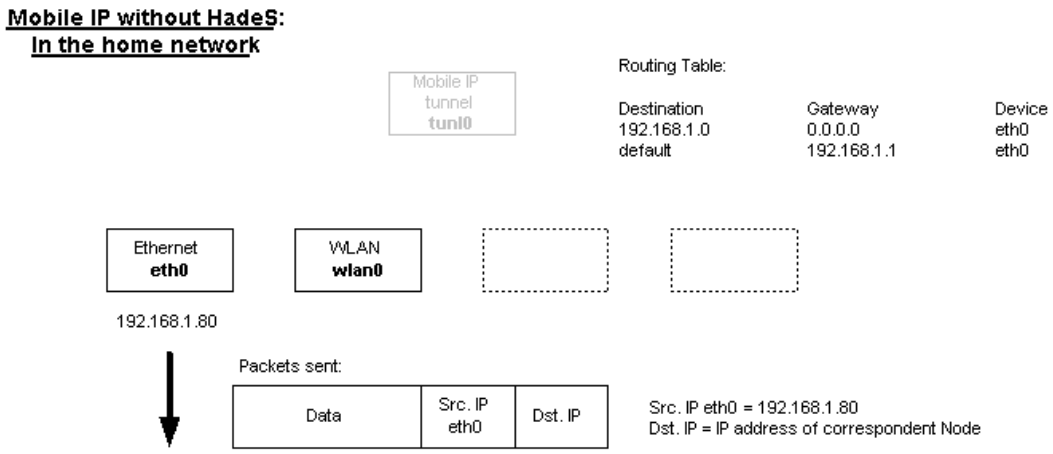


Figure 4.1 Mobile IP without HadeS in the home network

After being attached to a foreign network, a tunnel between the mobile node and its home agent is built. This tunnel device is built on the actual physical device (here the wireless device wlan0).

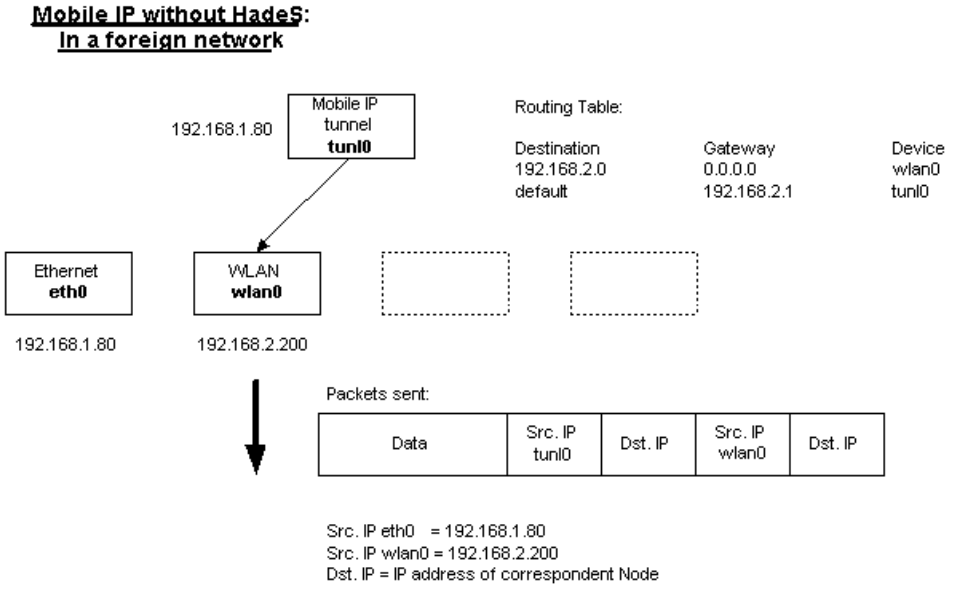


Figure 4.2 Mobile IP without HadeS in the foreign network

With HadeS as an intervening virtual device the routing looks different. Being at home, HadeS is configured with the home IP address. All communication passes through HadeS to the physical device.

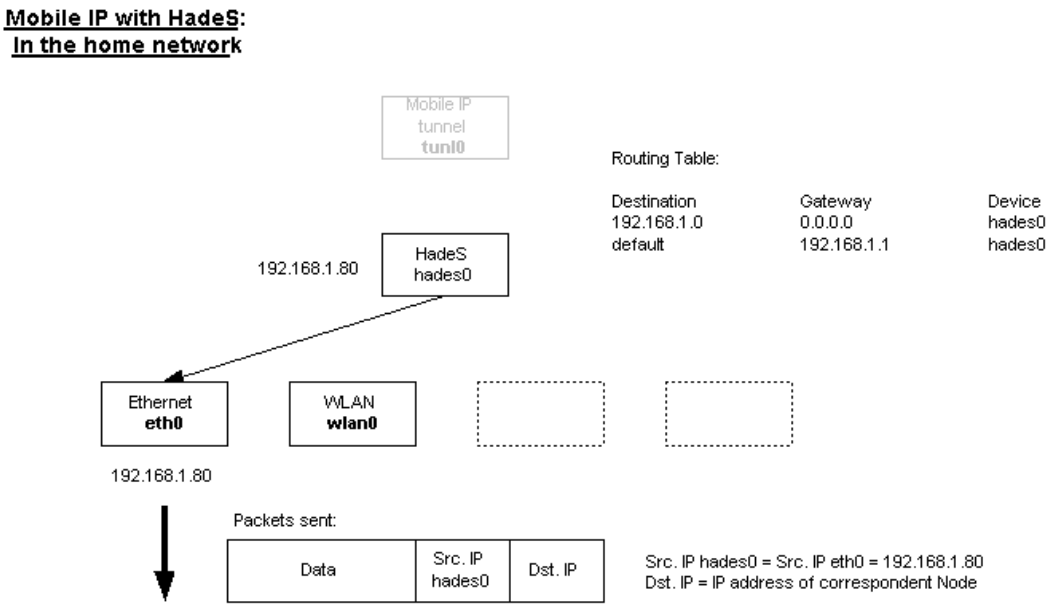


Figure 4.3 Mobile IP with HadeS in the home network

When changing access point, HadeS is linked to the new network interface and its IP configuration is updated according to the underlying device (see Figure 4.4; wlan0 with 192.168.2.200). The Mobile IP tunnel is now build on the virtual device HadeS to provide the home address to the applications.

The SecMIP module establishes a secured IPSec tunnel to the home firewall before sending the registration request to the home agent and creating the Mobile IP tunnel. This IPSec tunnel is built on HadeS.

As all the prototype was implemented under Linux, the step for the implementation of such a mechanism was to examine the possibilities of kernel programming. The first moments were really disappointing, searching for something totally unknown for us. After reading different books and taking contact with some "gurus" of the topic the solution seemed to be feasible. It was the starting point for the creation of a new Network Virtual Interface

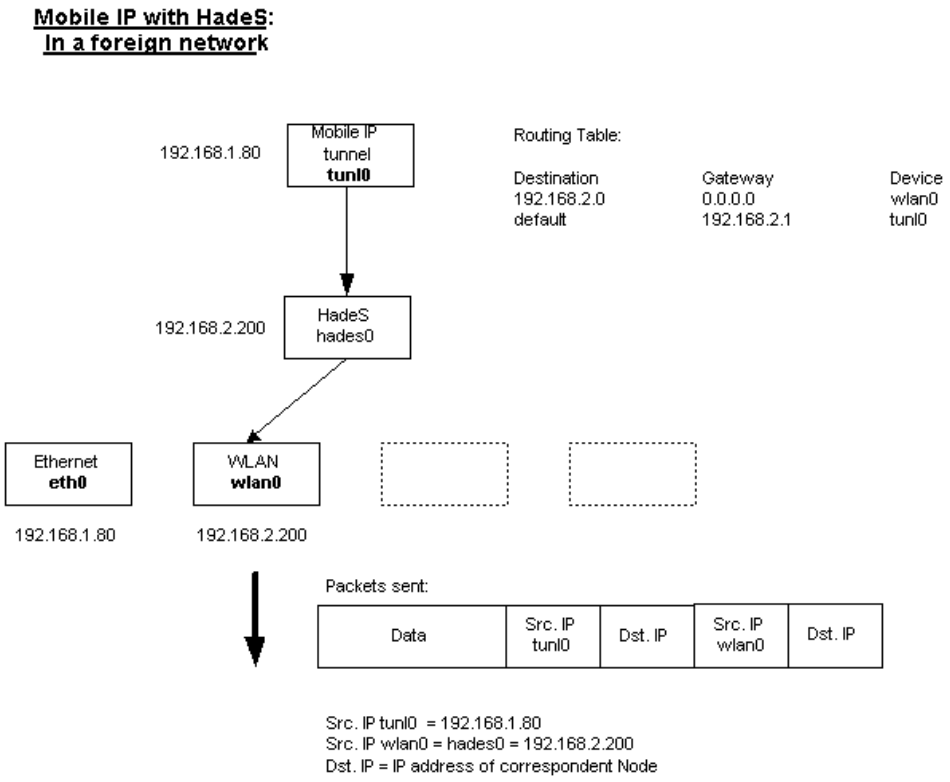


Figure 4.4 Mobile IP with HadeS in the foreign network

4.4 A FEW WORDS ABOUT THE LINUX KERNEL

4.4.1 PARTS OF THE KERNEL

In a Unix system, several concurrent processes attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resources. The kernel is the big chunk of executable code in charge of handling all such requests. Though the distinction between the different kernel tasks is not always clearly marked, the kernel´s role can be split, as shown in Figure 4.5, into the following parts:

4.4.1.1 PROCESS MANAGEMENT

The kernel is in charge of creating and destroying processes, and handling their connections to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality, and is also handled by the kernel. In addition, the scheduler, probably the most critical routine in the whole operating system, is part of process management. More generally, the kernel´s process management activity implements the abstraction of several processes on top of a single CPU.

4.4.1.2 MEMORY MANAGEMENT

The computer´s memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from simple *malloc*/*free* equivalents to much more exotic functionalities.

4.4.1.3 FILESYSTEMS

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top on unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, i.e. different ways of organizing data on the physical medium.

4.4.1.4 DEVICE CONTROL

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver.

The kernel must have embedded in it a device driver for every peripheral present on the system, from the hard drive to the keyboard and the tape streamer

4.4.1.5 NETWORKING

Networking must be managed by the operating system because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must correctly put to sleep and wake programs waiting for data from the network. Additionally, all the routing and address resolution issues are implemented within the kernel.

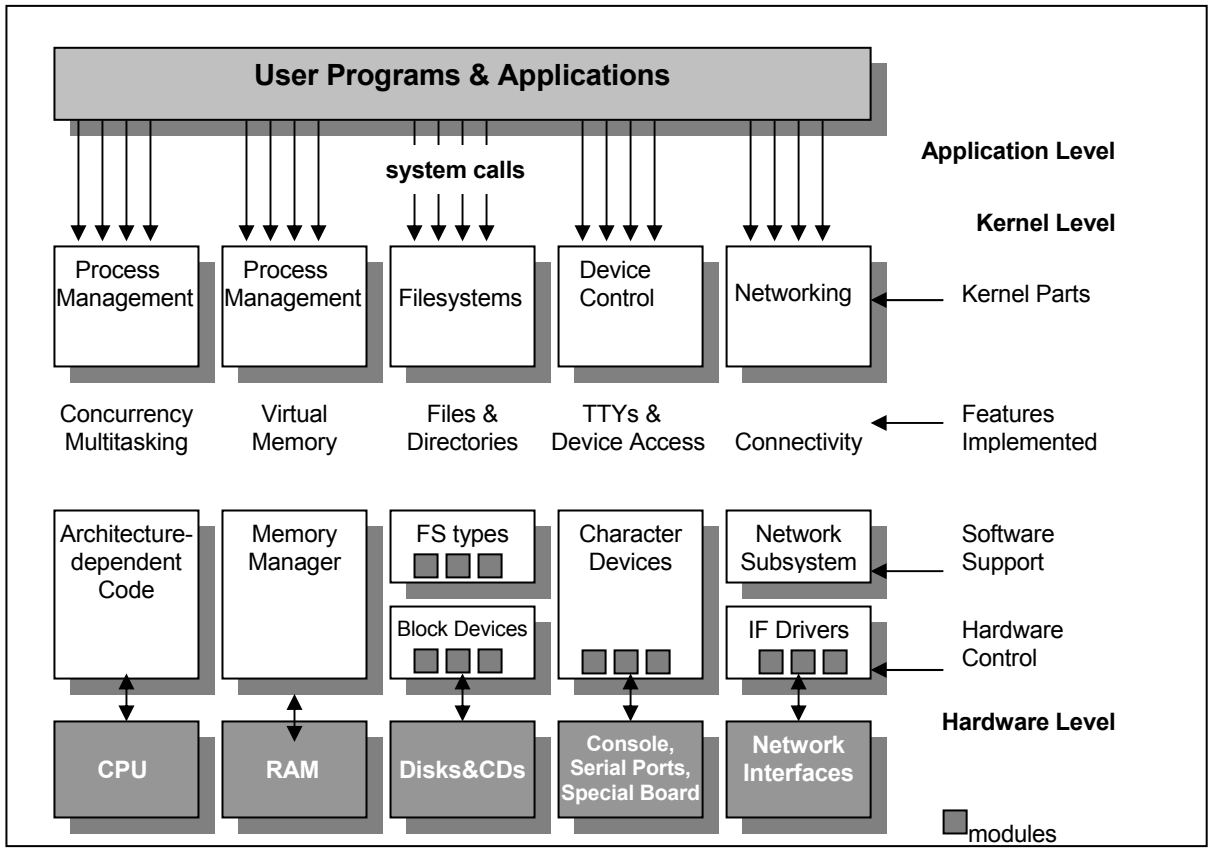


Figure 4.5 A split overview of the kernel

4.4.2 NETWORK INTERFACES

Any network transaction is made through an interface, i.e., a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a software tool, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both "telnet" and "ftp" connections are stream-oriented, they transmit using the same device; the device does not see the individual streams, but only the data packets.

Not being a stream-oriented device, a network interface isn´t easily mapped to a node in the filesystem, as */dev/tty1* is. The Unix way to call interfaces is by assigning a unique name to them (such as *eth0*). Such a name does not have a correspondent entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

There are other classes of "driver modules" in Linux but these are out of the scope of this project.

4.4.3 NETWORK DRIVERS

A network interface does not exist in the filesystem the way another Linux devices do. Instead, it deals with packet transmission and reception at the kernel level, without being bound to an open file in a process.

A network interface must register itself in specific data structures in order to be invoked when packets are exchanged with the outside world.

The network subsystem of the Linux kernel is designed to be completely protocol-independent. , This applies to both networking protocols (IP vs. IPX or other protocols) and hardware protocols (Ethernet vs. Token-Ring, etc.). Interaction between a network driver and the kernel properly deals with one network packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol.

In the next lines we go deeper in how the network interfaces fit in the rest of the Linux kernel, explaining the necessary commands and system calls to perform HadeS (Hardware Independent SecMIP device).

4.5 IMPLEMENTATION DETAILS: THE FIRST APPROCH

4.5.1 VIRTUAL NETWORK INTERFACES

In the Linux (or Unix) world, most network interfaces, such as *eth0* and *ppp0*, are associated with a physical device that is in charge of transmitting and receiving data packets. However, some logical network interfaces do not feature any physical packet transmission. Virtual devices are registered to the linux kernel as normal network devices. However they need the help of a real device to actually transmit packets. From the users and software engineers point of view a virtual device looks like a real device. It can be configured using the *ifconfig* command (e.g. assign a IP address) and routes can be configured to use it.

From the kernel's point of view, a network interface is a software object that can process outgoing packets, with the actual transmission mechanism hidden inside the interface driver. Even though most interfaces are associated with physical devices (or, for the loopback interface, to a software-only data loop), it is possible to design network-interface drivers that rely on other interfaces to perform actual packet transmission. The idea of a "virtual" interface can be useful to implement special-purpose processing on data packets while avoiding hacking the network subsystem of the kernel. The virtual interface is an additional tool for customizing network behavior.

4.5.2 HADES CODE

In this section we will describe briefly how HadeS was implemented. It is clear we will comment only the specific useful parts of the code for the good understanding of the complete design.

4.5.2.1 HOW AN INTERFACE PLUGS INTO THE KERNEL

Let us have a look at how this kind of interface attaches to the kernel and to the packet transmission mechanism.

The Linux kernel is already prepared to support different network devices. To integrate a new device it is only necessary to create a new device and initialize the new device object. When a new device is created it is added to the list of all network devices. In order to be able to support different network devices the Linux kernel provides a generic interface to the network devices. An upper layer (e.g. IP) which likes to transfer a packet just calls the transfer function of this interface. It is the interface's responsibility to pass the packet to the correct network interface. When a packet is received by the device driver some kind of protocol identifier is needed to demultiplex the packet to the correct upper layer protocol. In case of the Ethernet like protocol a type field is used.

Like many other kinds of device drivers, a network interface module connects to the rest of Linux by registering its own data structure within the kernel. The HadeS driver, for example, registers itself by calling *register_netdev*(&*HadeS_dev*);.

The device structure being registered, *HadeS_dev*, is a *struct net_device* object, and it must feature at least two valid fields: the interface name and a pointer to its initialization function:

```
...static struct net_device HadeS_dev = {
       name: "HadeS",
       init: HadeS_init,
   };
```

The init callback is meant for internal use by the driver: it usually fills other fields of the data structure with pointers to device methods, the functions that perform the real work during the interface's lifetime. When an interface driver is linked into the kernel (instead of being loaded as a module), the first task of the *init* function is to check whether the interface hardware is there.

The interface can be removed by calling *unregister_netdev()*, usually invoked by *cleanup_module()*. The *net_ device* structure includes, in addition to all the standardized fields, a "private" pointer (a *void **) that can be used by the driver for its own use. Where virtual interfaces are concerned, the private field is the best place to store configuration information; Listing One shows how the HadeS sample interface follows the good practice of allocating its own *priv* structure at initialization time.

{PRIVATE}*Listing One: HadeS Allocates Its Own priv Structure at Initialisation*

```
   /*priv is used to host the statistics, and packet dropping policy */
     dev->priv = kmalloc(sizeof(struct HadeS_private), GFP_USER);
     if (!dev->priv) return -ENOMEM;
     memset(dev->priv, 0, sizeof(struct HadeS_private));
```

The allocation is released at interface shutdown (i.e., when the module is removed from the kernel).

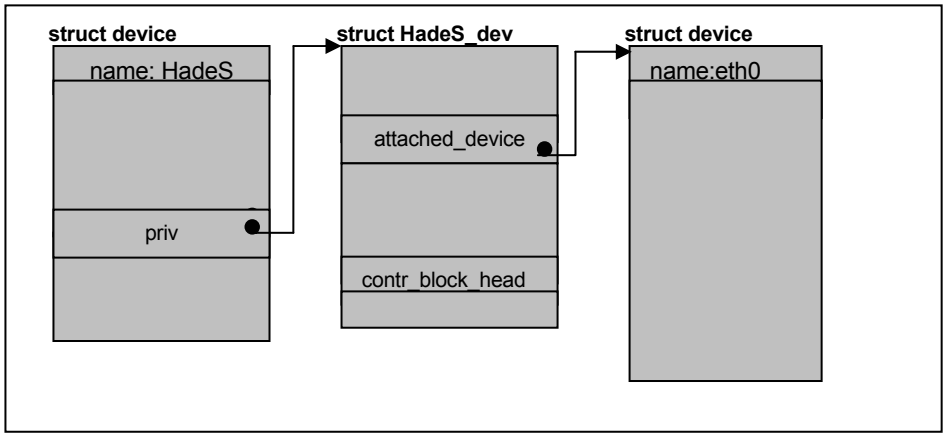


Figure 4.6 Integration of HadeS into the device list

4.5.2.2 DEVICE METHODS

A network interface object, like most kernel objects, exports a list of methods so the rest of the kernel can use it. These methods are function pointers located in fields of the object data structure, here *struct net_device*.

An interface can be perfectly functional by exporting just a subset of all the methods; the recommended minimum subset includes open, stop (i.e., *close*), *do_ioctl*, and *get_ stats*. These methods are directly related to system calls invoked by a user program (such as *ifconfig*). With the exception of *ioctl*, which needs some detailed discussion, their implementation is pretty trivial, and they turn out to be just a few lines of code (See Listing Two).

{PRIVATE} *Listing Two: HadeS allocates its own priv structure at initialisation*

```
int HadeS_open(struct net_device *dev)
{
   dev->start = 1;
   MOD_INC_USE_COUNT;
   return 0;
}

int HadeS_close(struct net_device *dev)
{
   dev->start = 0;
   MOD_DEC_USE_COUNT;
   return 0;
}
```

```
struct net_device_stats *HadeS_get_stats(struct net_device *dev)
{
   return &((struct HadeS_private *)dev->priv)->priv_stats;
}
```

The open method is called when you call *ifconfig HadeS up*, and close is called with *ifconfig HadeS down*; *get_stats* returns a pointer to the local statistics structure and is used by *ifconfig* as well as by the */proc* filesystem. The driver is responsible for filling the statistic information, whose fields are defined in *linux/netdevice.h*.

Other methods are related to the low-level details of packet transmission, but they fall outside of the scope of this discussion. The only interesting low-level method is *hard_ start_xmit*, which we discuss later.

4.5.2.3 IOCTL

The *do_ioctl* call is the most important entry point for virtual interfaces. When a user program configures the behavior of the interface, it invokes the *ioctl()* system call. This is how *shapecfg* defines network shaping and how *eql_enslave* attaches real interfaces to the load-balancing interface *eql* (*shapecfg*, *CBQ*, *eql*,..are some already implemented virtual interfaces). Similarly, the *HadESly* application configures the HadeS behavior on the HadeS virtual interface. Unlike "normal" device drivers, such as char and block drivers, the implementation of ioctl for interfaces is pretty well-defined. The invoking file descriptor must be a socket, the available commands are only *SIOCDEVPRIVATE* to *SIOCDEVPRIVATE+15*, and the infamous "third argument" of the system call is always a *struct ifreq** pointer instead of the generic *void** pointer. This restriction in *ioctl* arguments takes place because socket *ioctl* commands span several logical layers and several protocols.

The predefined values are reserved for a device's private use and are unique throughout the protocol stack. Note that no other ioctl command will be delivered to the network-interface method, so you really cannot choose your own values. Passing a predefined data structure to *ioctl* doesn't necessarily limit the flexibility of interface configuration, however, since the *ifreq* structure includes the data field, a *caddr_t* value that can point to arbitrary configuration information.

Based on the information above, the HadeS interface can be controlled using these commands (defined in *HadeS.h*):

```
#define SIOCHADESSETINFO SIOCDEVPRIVATE
#define SIOCHADESGETINFO
(SIOCDEVPRIVATE+1)
```

The actual use of the command within the user-space program *HadeSly* turns out to be pretty simple:

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
   /* a struct for passing data */
   struct HadeS_userinfo info;
   struct ifreq req;
   strcpy(req.ifr_name, "HadeS");
   req.ifr_data = (caddr_t)&info;
   /* fill info structure... */
   if ( ioctl(sock, SIOCHADESSETINFO,&req) < 0 ) {
      /* deal with error */
}
```

The kernel-space counterpart of the configuration process is slightly more complex, but only because it must deal with permission checks and copying data.

```
struct HadeS_userinfo info;
struct HadeS_userinfo *uptr;
/* check if authorized to set info */
if (cmd == SIOCHADESSETINFO && !capable(CAP_NET_ADMIN))
   return -EPERM;
/* get data from user space */
uptr = (struct HadeS_userinfo *)
   ifr->ifr_data;
err = copy_from_user(&info, uptr, sizeof(info));
if (err) return err;
/* ... use the info ... */
return 0;
```

4.5.2.4 PACKET TRANSMISSION

The most important tasks performed by network interfaces are data transmission and reception.

Whenever the kernel needs to transmit a data packet, it calls the *hard_start_transmit* method to put the data on an outgoing queue. Each packet handled by the kernel is contained in a socket buffer structure (*struct sk_buff*), whose definition is found in <*linux/skbuff.h*>. The structure gets its name from the Unix abstraction used to represent a network connection, the socket. Even if the interface has nothing to do with sockets, each network packet belongs to a socket in the higher network layers, and the input/output buffers of any socket are lists of *struct sk_buff* structures. The same *sk_buff* structure is used to host network throughout all the Linux network subsystems, but a socket buffer is just a packet as far as the interface is concerned.

A pointer to *sk_buff* is usually called *skb*.

The socket buffer is a complex structure, and the kernel offers a number of functions to act on it. The socket buffer passed to *hard_start_xmit* contains the physical packet, complete with the transmission-level headers. The interface does not need to modify the data being transmitted. *skb->data* points to the packet being transmitted, and *skb->len* is its length, in octets.

The most important entry point for a network interface driver is *hard_start_xmit*, where *hard* is short for *hardware*. This device method gets called whenever a network packet gets routed through the interface.

Where virtual interfaces are concerned, no actual hardware transmission takes place in the interface itself. The interface will instead resort to another network interface to perform transmission. Packet passing is implemented in two steps. First (usually at configuration time, within *ioctl*), the interface must connect to another interface, the one that can transmit packets. Next, its own *hard_start_xmit* must take proper action to pass the packet.

```
/* look for the hardware interface */
slave = __dev_get_by_name(info.name);
if (!slave) return -ENODEV;
priv->priv_device = slave;
/* .... */
/* update your statistic counters */
priv->priv_stats.tx_packets++;
priv->priv_stats.tx_bytes += skb->len;
/* give the packet to the hw interface */
```

```
skb->dev = priv->priv_device;
/* tell Linux to enqueue it */
dev_queue_xmit (skb);
```

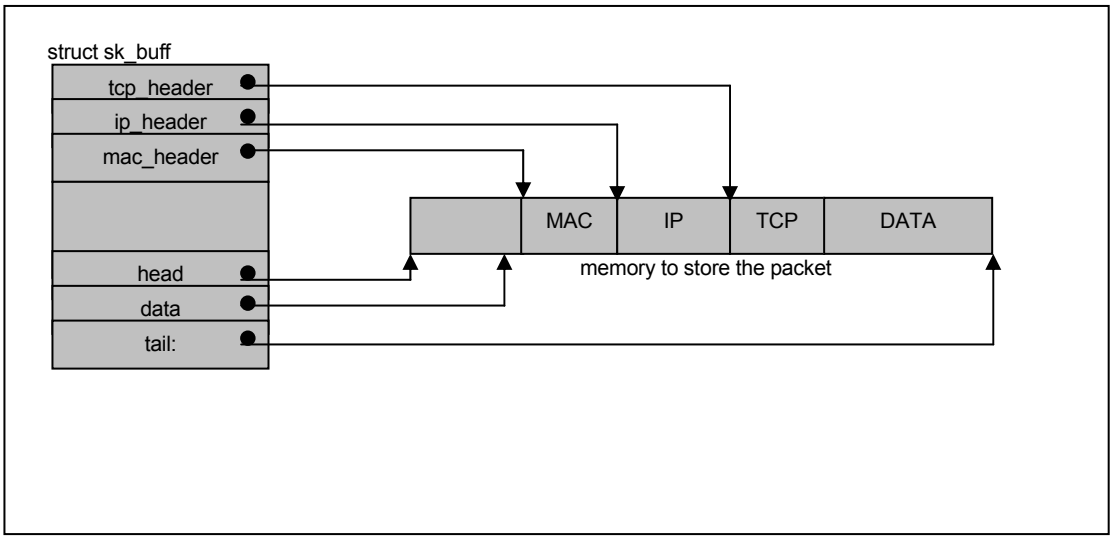


Figure 4.7 Linux sk_buff

Packets in Linux are stored in memory units which are called *sk_buff*. In the Linux operative system the packet is stored in a linear piece of memory. One member of the *sk_buff* holds a pointer of the network device which should be used to transmit the packet. This device pointer is obtained by the routing code. An abstraction of the *sk_buff* concept (how packets are stored) is shown in Figure 4.7.

In a perfect world, the virtual interface should also register a notifier call-back, so Linux will tell the driver when the physical hardware interface goes away. Though the first approach of HadeS is focused on the transmission so it does not register any call-back.

4.5.2.5 PACKET RECEPTION

Receiving data from the network is trickier than transmitting it because an *sk_buff* must be allocated and handed off to the upper layers from within an interrupt handler—the best way to receive a packet is through an interrupt since the vast majority of network interfaces is interrupt-driven.

When network packets hit an interface board, they generate an interrupt so that the operating system can handle packet arrival. (The only exception is the loopback interface, whose reception mechanism is part of packet transmission). A virtual interface, however, has no way to receive interrupts, and thus it cannot receive any network packets. This may seem unfortunate, because it would be nice to attach the same software operations to both directions of data flow.

Unfortunately, this is just not possible, and whoever needs to intercept incoming packets must use other ways to hook into the packets' path.

Proposal for solving the reception of packets via virtual network interfaces: each received packet is stored in a *sk_buff* and then passed to the generic device code. This determines the specific protocol which should deal with this packet by using the list of all registered devices and the protocol identifier. In our case the *sk_buff* should be passed to the receive function of HadeS. The processing of the received packet should not happen within the top-half of the interrupt handler but in the bottom-half. This guarantees that further packets can be received while a packet is processed. After HadeS function has finished its actions on the packet it passes the packet to the IP layer if protocol policy allows (e.g. only packet which are in order).

After different mail-brainstormings with Alessandro Rubbini, author of "Linux Device Drivers" [14], this implementation was shifted for later studies because of the complexity of such a task.

The right solution should instruct *eth0* to send stuff to HadeS instead of the default channel for incoming data. How packets are received from the hardware is handled by the Ethernet driver so a possible solution was to modify the behaviour of such a driver; for example calling *rx_function* (from HadeS) instead of calling *netif_rx* (kernel reception function).

Nevertheless this solution was rejected from the beginning because it did not cope with the expectatives of a scalable and network independent solution. The fact was that for every kind of card there was a different driver, thus the idea of changing driver per driver based on our requirements was totally unacceptable.

The figure below shows the attempt to solve the reception of packets via a virtual interface. Although it was not implemented, it served to understand better than before all the kernel processes related with transmission/reception.

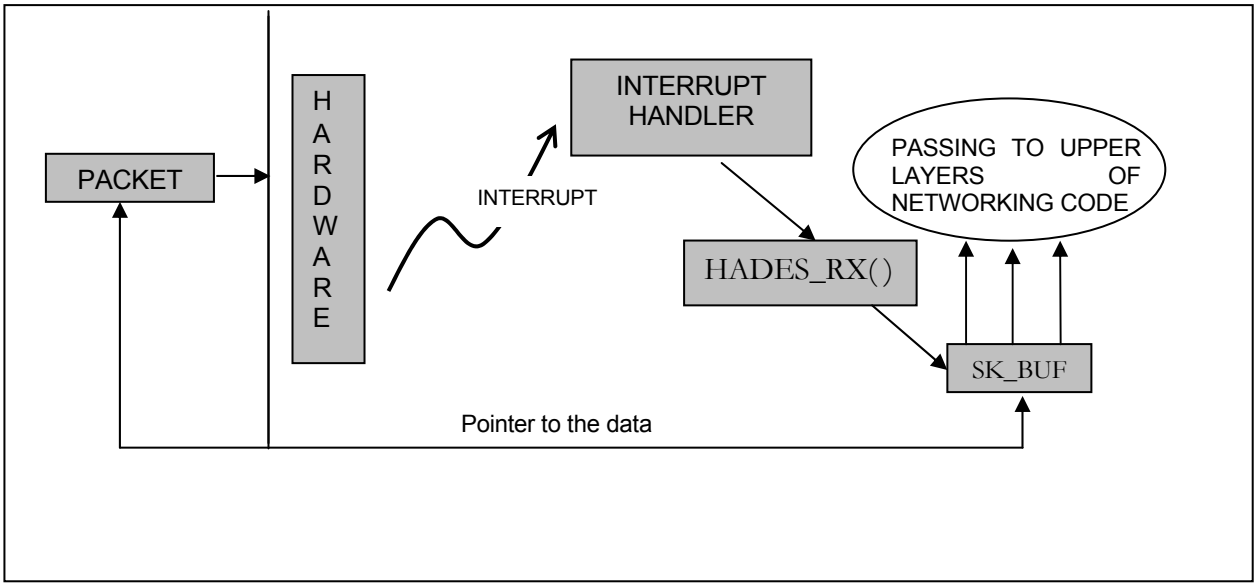


Figure 4.8 Packet reception process

4.6 USING HADES

All of this talking is rather pointless unless we can see the virtual interface at work.

The HadeS interface relies on an Ethernet interface for physical transmission, and it can be configured to operate in one of three HadeS modes. It can relay every packet (*pass* mode), relay only some percent of packets (*percent* mode, with an integer parameter), or turn relaying on and off on a repeated timely basis (*time* mode, with two parameters, on-time and off-time, specified as jiffy counts –architecture-dependent time quanta that correspond to 10 ms each for the PC platform). See Listing Three for an example.

{PRIVATE}*Listing Three: The three modes of HadeSly*

```
# HadeSly eth0 pass    ; # relay everything to eth0
# HadeSly eth0 percent 80    ; # drop 20% (pseudo random)
# HadeSly eth0 time 50 100    ; # relay for .5 seconds, drop for 1s
```

In order to connect HadeS to the network, you need to assign a local IP address to the interface (that IP address will be used as the source address, and remote hosts will use it to send their replies) and route some packets through it.

Current versions of Linux automatically associate a network route to each device, and this routing cannot be removed. Therefore, you can't reroute the entire LAN through HadeS at once. Listing Four re-routes a single host, called *Zurich*, in the routing table of the host *Bern*.

{PRIVATE} *Listing Four: Rerouting Morgana*

```
borea# insmod HadeS     ; # load module
borea# ifconfig HadeS Bern     ; # give same IP as eth0
borea# route add Zurich dev HadeS     ; # re-route this host
borea# ./HadeSly eth0 percent 60     ; # set dropping rate
```

With this setup, you can connect to *Zurich* with any protocol you like and experience a 40 percent packet loss. This loss is only on transmitted packets, unless *Zurich* runs another instance of HadeS with a similar configuration.

An interesting effect of this transmission path through two interfaces is that you can run *tcpdump* on both *eth0* and *HadeS* and see different results. While *tcpdump -i eth0* shows the packets being transmitted, *tcpdump -i HadeS* displays every packet sent out by the protocol layers before any dropping is applied. Figure 4.9 explains this behaviour by showing the path taken by a packet being transmitted through *HadeS*. Also the different device methods and functions used are drawn.

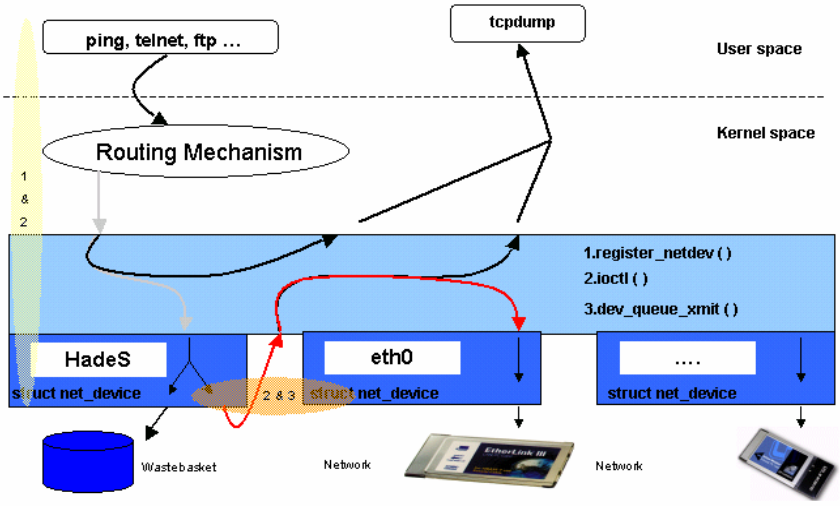


Figure 4.9 Data flow through HadeS

4.7 TESTS

The first testing purpose was to check the desired properties of our virtual interface. Summarizing we were interested in testing the next properties:

- Virtual interface directly linked to the physical interface
- Possibility to transmit packets through it
- Non-handling interruptions
- Permanently present at the routing table even removing the physical interface linked to it

In order to check these properties we cross-connected two Laptops, as shown in Figure 4.10.

With the help of the Linux command *"tcpdump"* we had the possibility to listen the packets between the different machines.

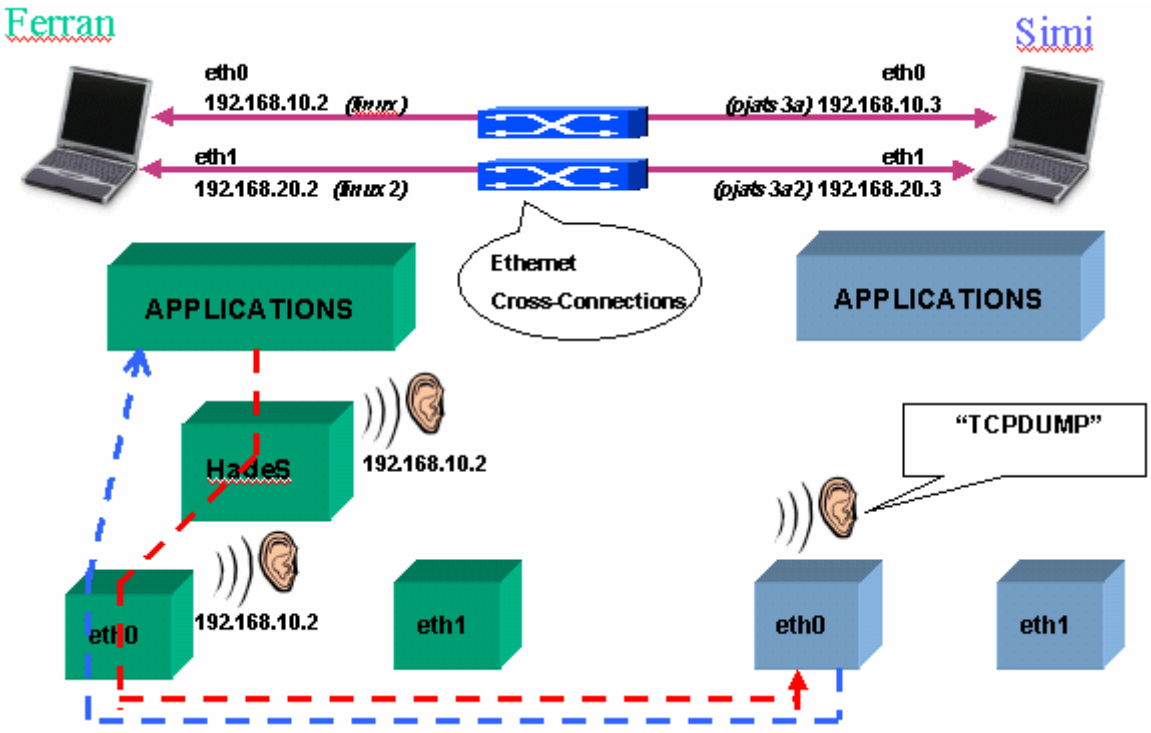


Figure 4.10 Testing HadeS

We tried to use the Linux command *"ping"* to send packets between the two machines and the different subnetworks. Pinging the Ferran´s Laptop from Simi´s Laptop we checked that effectively packets passed through HadeS and through eth0. With the next two commands we were able to see it:

```
tcpdump -x -i eth0

tcpdump -x -i  HadeS
```

So we checked the first two properties: all packets passing through *HadeS* go then through *eth0*, as a consequence we have transmission capacity for our virtual interface.

The next little step was to check if removing the physical interface linked directly to the virtual interface this one (HadeS) disappeared or no. We put it down (using *ifconfig* command) and even we unplugged the ethernet card; packets continued passing through HadeS. That meant applications (e.g., *tcpdump*) did not realize of any changes in the physical interfaces.

So we checked the two last properties: HadeS is permanently present even removing the physical interface directly linked to it because it does not support interruptions.

The figure below clarify the difference between using a normal Ethernet configuration or instead using a virtual interface to handle transmission, so avoiding interruptions and providing full mobility.

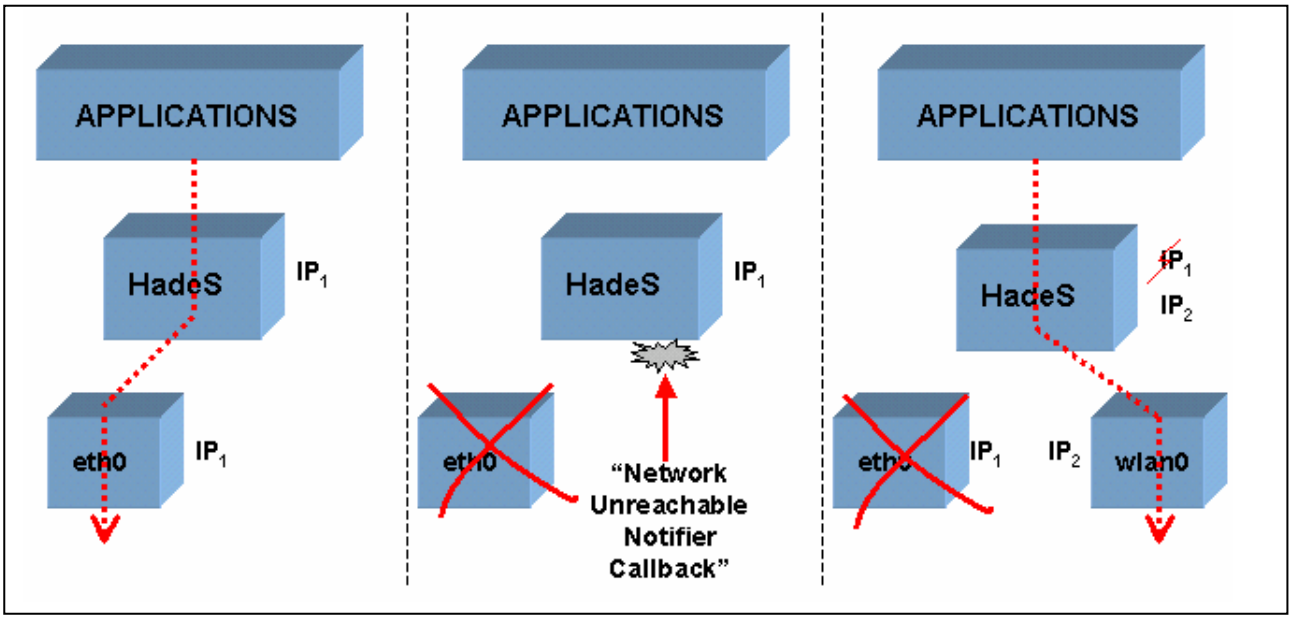


Figure 4.11 Applications do not crash using HadeS

4.8 SUMMARIZING

- HadeS is a virtual interface binded directly to the physical network device.
- Just changes its IP configuration when being linked to another physical network device.
- It makes the physical change of device transparent for the upper layers.
- "Virtual devices" can be useful to implement special-purpose processing on data packets while avoiding hacking the network subsystem of the kernel.
- The Hades implementation in "Advanced IP Mobility" is only a first approach. Open issues are still on the table and future attempts have to be done in order to reach the full mobility concept based on this project.

5 JORDI

5.1 OVERVIEW

The main purpose of JorDI (Jump On Right Device Intelligence) will be the management of the network interfaces based on the information it gets about the state of the link layer. This means mainly that if there is a break of connection JorDI has to change on an interface where a link layer connection is possible. As the Card Config Tool ensures that the interfaces will have connection on the IP layer JorDI can do his work without knowledge about the IP layer. When such a break will occur the application will not notice because HadeS avoids that the event will crash the application. With help of some Artificial Intelligence (the methods are not defined, yet) JorDI should be able to chose the best connection and it should also be able to determine a break of connection in advance in some cases so that it can try to establish another connection, so that the handover will then be much faster. All the decision are based on user given policies. Examples of reasonable policies:

- Always the fastest
- Always the cheapest
- Always the biggest possible coverage

This is only a suggestion and the JorDI implementation should be so flexible that it would be easily possible to define new policies, which will influence the behaviour of JorDI.

After such a change of interface has been performed by JorDI it has to update HadeS so that HadeS will change its binding on an the new interface and changes its IP address to the IP address of the new interface which is now the new care-of address. Then JorDI must inform the SecMIP implementation about the new situation so that the new tunnels can be created and the Mobile Nodes' connection can be established.

After this brief overview on how JorDI works, we can divide his functionality in three parts:

- Supervising the link layer
- Changing the connection
- Updating HadeS and informing SecMIP

5.2 SUPERVISING THE LINK LAYER

JorDI must all the time supervise the link layer of the inserted network devices. Based on the information JorDI gets from the link layer, on the information provided by the user in the Card Config Tool and on the chosen policy it must be able to maintain a ranking list of the network devices where always the best one is the connected one. Like mentioned JorDI has three sources of information: Card Config Tool database, policy and the link layer. The first two sources provide their information in an easy way, but what about link layer information? What information do we need?

5.2.1 ETHERNET

From network interfaces of this type we need to know about the state of the connection. This means whether the card is plugged or not. With this information we have all that we must know about the link layer.

Information needed from the driver:

- Connection state (cable plugged or cable unplugged)

Note: If there are more than one Ethernet devices it can be useful to know the throughput of the devices.

- Throughput

5.2.2 WIRELESS LAN

The important information needed from Wireless LAN devices is the signal strength between the MN and the Access Point (AP), it is attached to. With knowledge of this signal strength JorDI can decide whether there is a connection or not.

Information needed from the driver

- Signal strength

But with Wireless LANs it could be the case, that we have different APs with overlapping coverage (see Figure 5.1).

Figure 5.1 WLAN with different APs

In this case the signal strength to every AP must be known. But how can be detected which APs can be reached? Every AP sends out information at given time intervals. These packets are called beacons. JorDI must also be able to catch these beacons and to get the SSID out of it. We need this SSID to get a value for the signal strength to the given AP. With the knowledge of all reachable AP's SSIDs the signal strength for every AP can be determined and so JorDI can decide which AP it has to connect to. So, for Wireless LAN devices JorDI also has to maintain a ranking list of the reachable APs, with the signal strength as the ranking criteria.

The beacon format defined in the IEEE 802.11[15] Standard:

Order	**Information**
1	Timestamp
2	Beacon Interval
3	Cabability Information
4	SSID
5	Supported Frame Rates
6	FH Parameter Set
7	DS Parameter Set
8	CF Parameter Set
9	IBSS Parameter Set
10	TIM

Table 5.1 Advertised beacons

5.2.3 CONNECTION ORIENTED DEVICES

Connection oriented devices are most likely devices that are plugged to the serial or the USB port of a computer. The problem with these devices is that they can not be directly treated as network interfaces. They are referenced by the port device. Via this port device there must be first made a connection (most likely a dial-up to an ISP). Once this connection is established a network device will occur (e.g. *ppp0*). This newly obtained device is now the network interfaces through which we can access the network.

In this case JorDI should detect two kinds of link activities:

- physical link to the connection oriented device (e.g. the serial connection)
- link layer connection

5.3 CHANGING THE CONNECTION

This is the part where JorDI does change the connection. Like seen in the overview JorDI will have to maintain a ranking list of the best fitting device. For Wireless LAN devices it also has to maintain a ranking list for the AP which it has the best signal strength with. (We focus here on the signal strength. There are other criteria that can be treated like QoS. But this goes beyond the focus of this project.) These two tables are build on the information we mentioned in the previous chapter and they have to be permanently up to date. So JorDI will always force the best connection to be used. Once the first place in the ranking changes, JorDI has to change the used card. It would be important to implement some test routines to prevent that JorDI does to many switches (e.g. when we are just moving between two APs and the signal strength changes too often).

When we concentrate on the three most important network interface types, which are Ethernet, Wireless LAN and serial devices, then we have the following possible scenarios:

- Wireless – Wireless

  This handover is by far the most important for our purpose. We must assume that when we are moving we have to change from one Wireless LAN AP to another. This is unavoidable in order to get a high coverage. To communicate with an AP, to get the needed care-of address, we must have the SSID and the WEP keys, if encryption is enabeled. We can assume that mostly this encryption is not enabled since we intent to use public APs. Like mentioned in the previous section this information we can get out of the advertised beacons.

  JorDI has to perform a change from one AP to another when the first place in the ranking list for the Wireless LAN changes. This must be done with care, because the rankings in this list will change very often. That fore reasonable thresholds must be defined, what can be done when there is a testbed to collect more statistical values.

- Wireless – Ethernet

  This will take place as soon JorDI detects, that a link to an Ethernet network is available. There should be the possibility to stay on Wireless, for example when the user plans to re-disconnect immediately.

- Ethernet – Wireless

This case is clear, when we lose connection to the wired network and a Wireless one is available, JorDI will invoke that change.

- Wireless – Serial

With this handover the user will be asked to make a dial-up connection. In a wireless network we can with the help of the signal strength estimate when we will lose the connection to any AP. With help of this estimation JorDI should invoke a dial-up to get the network device so it can shorten the time when no connection is available. A handover on a serial device should never be made without user agreement, cause it is most likely the most expensive alternative.

- Serial – Wireless

For this case we have to use the beacons again. As soon as a wireless connection is available we will switch!

- Ethernet – Ethernet

This handover will not occur very often. But in case it should be performed properly the tool must know about the link layer state of the to devices. As soon as we lose the connection on one Ethernet device we will switch to the other. Or when a faster one is available (switch from 10 Mbits interface to 100 Mbits interface). In this case it is useful to obtain the throughput of the two devices. This we can take out of the configuration database, if it is not measurable. But then it is a user defined value the user specifies at configuration time with help of the wizard of the Card Config Tool.

- Ethernet – Serial

In this case the handover will be performed as soon as we lose the connection with the Ethernet device. A dial-up will be done and the SecMIP updated. This is the worst case. This can cause a big delay as the tool can not guess when an Ethernet connection will break. Therefore it should be possible for the user to do the dial-up at any time and to perform this handover manually when he plans to move, so that the handover delay can be decreased.

- Serial – Ethernet

This is quite clear. As soon as possible. Ethernet is much faster and much cheaper.

- Serial – Serial

Here we have to know the kind of serial devices we are using. Then the tool has to decide with help of some user defined values in the Card Config Tool database and the chosen policiy which one will be used. The criteria can be the bandwidth or the costs.

5.4 UPDATING HADES AND INFORMING SECMIP

Finally JorDI has to inform the other tools what is happening. It has to update HadeS as soon as there is a new care-of address available and to inform SecMIP about this new address. Updating HadeS is rather easy an can be performed with the following command:

`HadeSly eth0 pass` (where eth0 will be the name of the actually active device)

After having updated HadeS JorDI must inform SecMIP about the new care-of address. The daemon of SecMIP should offer an interface for this purpose. With help of this interface JorDI is able to provoke a SecMIP update on the new care-of address of the device it just changed to.

5.5 CONCLUSIONS

JorDI is a suggestion purposed when the project was divided in three main parts. The studies in this topic have just begun. There are a lot of issues that still have to be treated and a lot of experience that has to be collected.

JorDI is the most important, the most complicated and most time consuming part of this project. It has never become real but this chapter documents the idea that is behind this part of JorDI.

6 OUTLOOK

6.1 CARD CONFIG TOOL

The Card Config Tool is still in a development phase. There are a lot of enhancements that can be done to make this tool useful for daily work. It is not very stable yet, but it already works fine for our prototyping purpose. This chapter is purposed to give an overview of work that still is to be done for this incomplete implementation.

6.1.1 SCHEMES

Like mentioned in the section of the Card Config Tool SuSE Linux provides a mechanism which allows to define different configuration schemes for the given network devices. With these schemes one can define a configuration for a device which works well in the office an another one which works at home. These schemes are treated in this implementation only in an indirect manner: The user can give a string for a scheme. This scheme is then written to the configuration file. When the cardmgr of the PCMCIA package invokes the appropriate script, the configuration for the currently selected scheme will be used. So it is possible to define different schemes, but these schemes are not managed, and these schemes still have to be selected by the following command:

`cardctl scheme SuSE` (where SuSE is the name of the scheme)

The enhancement could now be to have an internal list of all defined schemes and to make them available in a selection, so the user can specify the wished scheme in the GUI

6.1.2 INSTALLATION

A little HOWTO was written to help with the Installation of this implementation of the Card Config Tool. This process is not very comfortable and not user friendly at all. It would be possible to automate the steps of this installation guide so the user does not have to be experienced to install this tool.

6.1.3 THE GUI

For our purpose the GUI did not turn out to be the most important part of the implementation. Some time was spent on it but it was not the focus. So the GUI is useful and nice, but not very well implemented. For further implementation it is unavoidable to re-engineer the GUI that it becomes scalable. In this shape the GUI can only be heavily scaled and maintained.

6.1.4 NEW TECHNOLOGIES

Apart from the GUI the implementation of the Card Config Tool follows the principles of an OOP approach which makes it easily scalable. It is that for not very complicated to add new technologies in this implementation. The next step should be the integration of GPRS technology.

6.1.5 NON PCMCIA DEVICES

This is the most important step for the next phase of the implementation. Even if in laptop most of the network devices are PCMCIA devices there are a lot of built-in cards in Mobile Computers. In this release the Card Config Tool is not able to recognize those devices too. In the next step this fact should be treated to have a complete range of supported devices.

6.2 HADES

Like mentioned in the section of HadeS, packet reception is the most delicate point. The decision must be made if it is important to receive packets through this virtual device or not. When the virtual device must have this ability then the drivers of the different devices must be changed to invoke the reception function of HadeS instead of the one of the kernel.

To resolve this problems there could also be another solution. It should be possible to change the kernel reception function in the manner that it invokes the reception function of Hades and the one of HadeS then processes the packets in his own way.

Figure 6.1 shows the normal way how packets can be received. The driver invokes the right reception function which will be responsible to handle the packets out for further processing of them. So the driver must call the normal kernel function or the one of HadeS. If for every driver HadeS should receive the packets then every driver must be changed so that it call the function of HadeS.

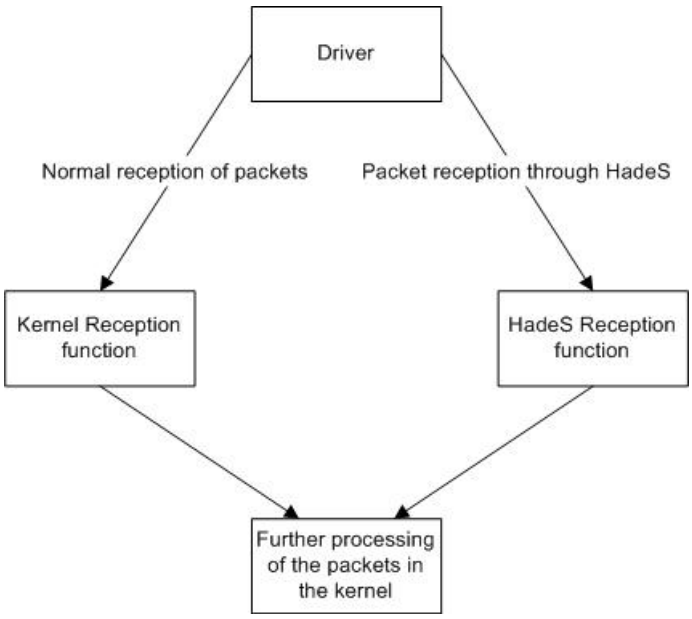


Figure 6.1 Normel packet reception process

We can solve this problem in the following way:

- Change the name of the original packet reception function of the kernel (let's say reception2)
- Then we have to write a new function with the name of the original kernel reception function which we simply invoke the reception function of HadeS
- Now the packets can be processed by the HadeS reception function

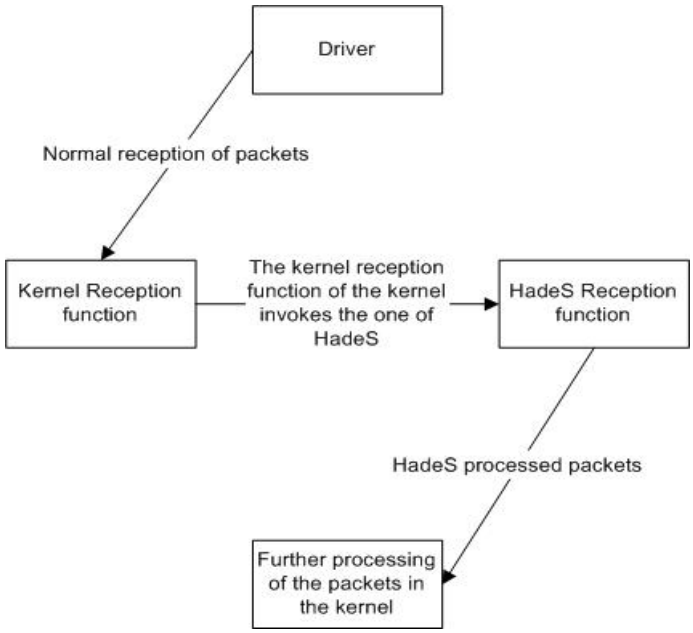


Figure 6.2 Packet reception through HadeS with modified kernel

6.3 JORDI

For JorDI there is not much to say left as the documentation of JorDI already is like an outlook. But one thing still has to be mentioned in this part, the problem with the connection oriented devices.

The lack of experiences in this topic does make it impossible to define what information we exactly need from the driver to provide JorDI's functionality. So deeper studies are needed to gain the acknowledge how this kind of devices are managed by the kernel and how the problem of this devices can be solved.

A great importance will also be to study how we can perform a handover from one AP to another. Like one can see above, it is important that JorDI can provide some mechanisms to avoid too many changes as signal strengths can change very frequently. To resolve this problem it would be a good idea to have a look at the cellular protocols like GSM to get some inspiration how the handover related problems can be solved in a cellular network and how these ideas can be adapted on the Wireless LAN.

The other requirements are pretty clear, but not the way how we can get these out of the current driver implementations. Maybe it would be necessary to change the drivers to get these parameters.

Last but not least, we must come back to the topic of GPRS. This technology is now very new and ISPs just begin to offer this service. For that reason we do not know exactly how GPRS will work. Now there are to possibilities, a connection-oriented and a connection-less. This problem is not only important for the technique of a handover but also for the cost related aspects (do we have to pay for the traffic or for the online time?).

Projects in the topics of Wireless LAN handovers and GPRS are defined at CT and shortly there are new students which will begin to work in these topics. So new know-how and experience will can be collected to resolve these problems.

7 RELATED WORK

7.1 FUZZY LOGIC FOR A HETEROGENEOUS IP ENVIRONMENT

[16]

7.1.1 INTRODUCTION

This document describes a handover mechanism for heterogeneous networks. It treats the three following technologies: UMTS, GPRS and satellite communication. But should be easily adoptable to a more complex scenarios with Ethernet and WirelessLAN in addition to the three technologies above. Here will only be shown a brief introduction to the idea. The interested reader can refer to [16] for a more detailed explanation of the fuzzy-logic concept for handover decisions.

7.1.2 HANDOVER MANAGEMENT

In homogeneous networks, the most common reason for initiating a handover occurs when the carrier-to-interference (C/I) ratio or received power falls below a specified threshold. In this section a new handover procedure is proposed to support vertical handover between heterogeneous networks. This is achieved by incorporating the Mobile IP principles in combination with fuzzy logic concepts utilizing different handover parameters. In Mobile IP, when the mobile node is not attached to the home network, the HA receives all the packets destined for the node and forwards them to the node's current point of attachment. This feature makes Mobile IP suitable for handling intersegment (vertical) handover or macromobility. In such cases, the mobile node obtains a home address from its home network and only changes its CoA when the active segment changes. If an intrasegment handover occurs in the same segment, the CoA remains the same. Handover can normally be separated into three phases: initiation, decision, and execution. In this article, the fuzzy logic concept is applied to the initiation and decision phases. During handover initiation, the T-IWU[1] collects information on the user profile, QoS perceived by the user, and radio link availability. Based on this information, together with information on segment availability, a handover can be initiated. In the handover decision phase, the T-IWU is responsible for selecting the most suitable target segment. The selection of the segment depends mostly on the user profile. A standard user profile contains information such as the minimum and maximum cost and the list of segments with the highest and lowest priority. During handover execution, two main procedures are performed. The first procedure is used to establish IP connectivity to the core network of the access segment. In other words, the terminal is required to obtain an IP address from the selected segment. This is

[1] The T-IWU is a part of a Multi Mode Terminal (MMT) which is capable to work in all access segments (UMTS, GPRS and satellite)

known as PDP context activation in terrestrial mobile systems, such as GPRS. Upon completion of this procedure, standard Mobile IP v. 6 (MIPv6) procedures are executed to register the new CoA to the HA. At this stage, the terminal will have two active IP addresses, since a mobile node may use more than one CoA at a time. In this case, the terminal will have a primary CoA and a secondary CoA. Once the new CoA has been established, the old CoA should still be used as a secondary CoA for a specified time. This is to reduce packet loss in the system; ensuring handover is as smooth as possible.

7.1.3 APPLICATION OF THE FUZZY-LOGIC-BASED HANDOVER CONCEPT

The proposed solution to initiate handover utilizes the fuzzy logic concept, where a robust mathematical framework for dealing with imprecision and nonstatistical uncertainty is introduced
For handover initiation, four different criteria are used: bit error rate; network coverage; perceived QoS; and signal strength (SS). This could be changed depending on the requirements of the system.
For handover initiation, the algorithm is separated into three different stages.
In the first stage, the parameters of the system are fed into a fuzzifier, which will transform the real-time measurements into fuzzy sets.
The second stage involves feeding the fuzzy sets into an inference engine, where a set of fuzzy rules is applied. Fuzzy rules can be defined as a set of possible scenarios utilizing a series of IF-THEN rules, which decides whether handover is necessary. Following this, a set of different handover decisions can be obtained. The decision set could be classed into four different sets; yes (Y), probably yes (PY), probably no (PN), and no (N). of the user.
However, since a measurement usually falls into more than one category, more than one decision set can be obtained following the implementation of the IF-THEN rule. At this stage, it is useful to convert the resultant fuzzy decision sets into a precise quantity. This is implemented in the third stage, which is also known as defuzzification. Here, the membership values and decision sets are used to obtain the handover factor using a method known as the *centroid method.*[16]
When considering the handover decision algorithm, inputs from both the system and the user are required. The main purpose of the handover decision algorithm is to select a segment for a particular service that can satisfy the following objectives: low cost, good signal strength, optimum bandwidth, low network latency, high reliability, and long battery life, while taking into account the preferred segment of the user. There are two different stages in the handover decision algorithm.
The fuzzy ordinal ranking procedures a classical/crisp set, which is used to influence the weighting of the criteria. The crisp set consists of information on segment availability, terminal type, and any segments prohibited from use.

8 ACKNOWLEDGEMENTS

The work on this project was a really interesting experience for me. I have learned that it is not that simple to provide the functionality of a automated handover. But I am still convinced that we have done a good job and it was useful for further studies in that topic.

I would like to thank Ferran Moreno Blanca for his cooperation. It was an inspiration to exchange the knowledge and to gain these experiences in team working.

I would also thank Marc Danzeisen for his faith in my capabilities and for his care when I was stuck in my approach.

Last but not least my thanks go to Jan Linder, who had always some minutes even when he was stressed. I would like to thank him for all the organization and for helping me to find my way in such a big company like Swisscom AG.

9 REFERENCES

[1] Ferran Moreno Blanca, „Advanced IP Mobility Solution“, Diploma work of the Universitat Politècnica de Catalunya (UPC), Barcelona, September 2001,

[2] Marc Danzeisen, “Secure Mobile IP Communacation”, Diploma work of the University of Bern, May 2001, http://www.iam.unibe.ch/~rvs/publications/SecMIP.pdf

[3] V.Gupta, G. Montenegro, “Secure and mobile Networking, Mobile Networks and Applications 3 (381-390)”, Balzer Science Publisher BV, 1998

[4] C. Perkins, “IP Mobility Support”, October 1996, rfc2002.txt

[5] R. Droms, “Dynamic Host Configuration Protocol”, March 1997, rfc2131.txt

[6] Marc Danzeisen, „Access of Mobile IP Users to Firewall protected VPNs”, http://www.iam.unibe.ch/~rvs/publications/secmip_gi.pdf

[7] General Linux website, http://www.linux.org

[8] S.u.S.E. GmbH home page, http://www.suse.com

[9] The Python project web site, http://www.fsbassociates.com/books/pythonchpt1.htm

[10] The GTK project web site, http://www.gtk.org

[11] GTK tutorial, http://www.gtk.org/tutorial/ch-introduction.html

[12] The Gnome project web site, http://glade.gnome.org/

[13] Open Source development hosting project http://glc.sourceforge.net/

[14] Alessandro Rubini, “Linux device drivers”, O’Reilly 1998

[15] IEEE P802.11, The working group for Wireless LANs, http://grouper.ieee.org/groups/802/11/

[16] P. M. L. Chan, R. E. Sheriff, and Y. F. Hu, University of Bradford, United Kingdom
P Conforto and C. Tocci, Alenia Spazio, Italy, “Mobility Management Incorporating Fuzzy Logic for a Heterogeneous IP Environment”