INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Combining Heterogeneous Access Networks with Ad-Hoc Networks for Cost-Effective Connectivity

João Miguel Marques de Almeida da Cunha Mota

Dissertação para obtenção do Grau de Mestre em

Engenharia de Redes de Comunicações

Júri

Presidente: Prof. Dr. Paulo Jorge Pires Ferreira
Orientador: Prof. Dr. Artur Miguel do Amaral Arsénio
Co-Orientadora: Prof. Dra. Helena Rute Esteves Carvalho Sofia
Vogal: Prof. Dr. Rui Jorge Morais Tomaz Valadas

Junho 2011

Agradecimentos

From the formative stages of this thesis, to the final version, I owe an immense debt of gratitude to both my advisers, Prof. Dr. Artur Arsénio and Prof. Dr. Rute Sofia, who were abundantly helpful and offered invaluable assistance, support and guidance. Their sound advices and careful guidance were invaluable during the course of the whole work.

I would also like to thank my family, who offered me unconditional love and support throughout the course of this thesis.

Lisbon, June 2011
João Mota

Resumo

Devido à proliferação de tecnologias sem fios de baixo alcance tais como *Wireless Fidelity* (Wi-Fi) ou Bluetooth os dispositivos móveis com capacidades multihoming estão a proliferar. A existência de diferentes interfaces físicas nestes dispositivos torna-os capazes de se interligarem a diferentes redes heterogéneas de uma forma auto-organizativa. Actualmente, para tornar as redes ad-hoc mais confiáveis, têm-se utilizado técnicas como o multihoming e o balanceamento de carga. No entanto, este tipo de técnicas não utiliza de uma forma eficiente e simultaneamente todas as interfaces físicas de rede presentes nos dispositivos móveis.

Esta tese aborda o tema da utilização simultânea das várias interfaces sem fios de um mesmo dispositivo, tendo como objectivo principal analisar técnicas que possam permitir um melhor desempenho da rede. A análise deste desempenho assenta numa análise do melhoramento do débito e latência da rede. Para tal foi implementada uma abstracção situada entre a camada de aplicação e as várias interfaces de rede, que pode ser usada em redes ad-hoc heterogéneas.

Esta solução tem por base uma interface virtual que permite o uso simultâneo de várias interfaces de rede, escondendo a heterogeneidade das aplicações, e que permite a adição de um qualquer número de interfaces de rede, aumentando assim o ritmo de transmissão total do dispositivo.

Abstract

With the advent of modern technology, mobile devices with multihomed capabilities are proliferating. Existence of different network interfaces in multihomed devices gives them the possibility to explore seamlessly roaming across heterogeneous networks. To make ad-hoc networks more reliable, one has often to use techniques such as multihoming and load-balancing. However, these techniques do not make full use of all network interfaces presented in a mobile device.

This thesis addresses the topic of using multiple network interfaces simultaneously as a way to increase the available throughput in wireless networks. It studies and compares different techniques that have been previously presented in the literature, and proposes an architecture applicable to a broader range of networks. To do so, we have implemented an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc networks, from a mobile node (station) perspective. By heterogeneous it is here meant networks where nodes can transmit by relying on several short-range wireless technologies.

Our solution is based on a *virtual interface* (vi) approach, which allows the usage of all active interfaces of a mobile device simultaneously, while hiding the heterogeneity from the applications and allowing any number of interfaces to be added, in the expectation of increasing the overall wireless throughput.

Palavras-chave
Keywords

Palavras-chave

Multihoming

Redes sem fios heterogéneas

Eficiência

Interface Virtual

Balanceamento de carga

Keywords

Multihoming

Heterogeneous wireless networks

Efficiency

Virtual Interface

Load-Balancing

Index

List of Figures

List of Tables

List of Graphs

Acronyms

PDA	Personal Digital Assistant
GPRS	General Packet Radio Service
UMTS	Universal Mobile Telecommunication System
AP	Access Point
UE	User Equipment
Vi	Virtual Interface
ABR	Available Bit Rate
RTT	Round-trip time
MAC	Media Access Control
WPAN	Wireless Personal Area Network
MANET	Mobile Ad-hoc Network
AODV	Ad-hoc On-Demand Distance Vector
OLSR	Optimized Link State Routing Protocol
PAN	Personal Area Network
EMF	End-to-end Mobility management Framework
UHCI	Universal Host Controller Interface
GPL	General Public License
PCI	Peripheral Component Interconnect
MTU	Maximum Transmission Unit

1 Introduction

The transparent support of a multitude and variety of existing and emerging wireless and wired networking technologies is a driving force towards convergence of networks. Moreover, it is commonplace nowadays to have electronic devices with multiple networking capabilities. Personal computing devices, e.g., laptops, PDAs, smartphones, are typically equipped with several networking interfaces ranging from different flavours of Wireless *Fidelity* (Wi-Fi) to Ethernet, GPRS, UMTS, and Bluetooth.

Adding to the diversity of network interfaces that end-user devices today include, the common Internet end-user has at his/her disposal a set of applications with significantly different bandwidth requirements and which comprise multimedia services, gaming, as well as collaboration, among others. However, most services provided today to the end-user simply take advantage of one network interface at a time.

This perspective is bound to change due to the fact that more and more, different *Service Providers* (SP) serve the same household or enterprise location. As an answer to this increasing complexity, several traffic-engineering techniques are being applied to take advantage of the different interfaces available on a single device. This is the case of multihoming (cf. section 2.3.1) and load-balancing (cf. section 2.3.2), techniques which have been used to give networks some redundancy and redirect traffic flows based on the device necessities (power, signal strength, available bit rate, etc), thus assisting in making the network more robust. Hence, multihoming and load-balancing aspects are to be surveyed, analyzed and compared to the work developed in this thesis, but as will be seen, the multiple and simultaneous use of different interfaces is still in an embryonic state, since it is not yet possible to make full use of all the physical interfaces present in mobile devices.

Our main objectives are two-fold. Firstly, to understand up to which point and for which cases it is relevant to consider a single interface (as a virtual container for all the potential network interfaces in an end-user device). Secondly, to analyze and evaluate up to which point is possible to achieve an efficient utilization of multiple network interfaces by devices via rate control and optimal assignment of traffic flows to available networks.

The remainder of this document is organized as follows:

Chapter 2 surveys previous work in this area, addressing several possible ways to improve the effectiveness of a heterogeneous ad-hoc network, as well as some problems that may arise from the implementation of such solutions.

Chapter 3 describes the model of the proposed solution followed by the implemented architecture and approach taken, that transparently improves the cost-effective connectivity in heterogeneous access networks.

Chapter 4 introduces some goals and the methodology followed. A generic description of the evaluation parameters and scenarios is then provided, followed by a description of the topologies implemented and of traffic settings. Finally it explains in a detailed manner the results obtained during the experiments.

Chapter 5 concludes the thesis and proposes some directions for future work.

1.1 Generic Applicability Scenarios

This section provides an overview on global applicability scenarios that are the basis of the functionality to be described. Let us first provide a hypothetical scenario. Imagine a user in an enterprise setting participating in a video conference call via his/her multihomed[1] device, which incorporates both a Bluetooth and a Wi-Fi (e.g. IEEE 802.11g) interface. While engaged in the conference proceedings, the user is uploading content on a remote server for the participants to access, and at the same time needs to retrieve some files from the server. Data is transmitted by the device which dynamically monitors the interfaces at its disposal. The device then routes the traffic via these physical interfaces based on the varying network characteristics like *Available Bit Rate* (ABR), delay and signal strength, and also based on specific user expectations (e.g. increase energy savings). By doing this the device would be able to use the interfaces at its disposal, but would not offer a transparent solution for the remaining of the network, and would not be using the interfaces at full capacity, since it is only allocating the data through them.

What we propose can be seen in Figure 1.1, it provides technical details concerning one potential applicability scenario, where different multihomed end-user devices are interconnected in an ad-hoc way. Specifically, we consider four mobile devices, three of which are multihomed (Bluetooth and Wi-Fi interfaces). The physical interfaces of each mobile device are integrated into a global, virtual interface, represent on OSI Layer 2 by a virtual MAC, and on OSI Layer 3 by a virtual IP, where the physical interfaces are behind the virtual interface, offering a transparent solution for both the application layer and the remaining of the network.

[1] Multihomed describes a computer host that has multiple IP addresses to connected networks. A multihomed host is physically connected to multiple data links that can be on the same or different networks.

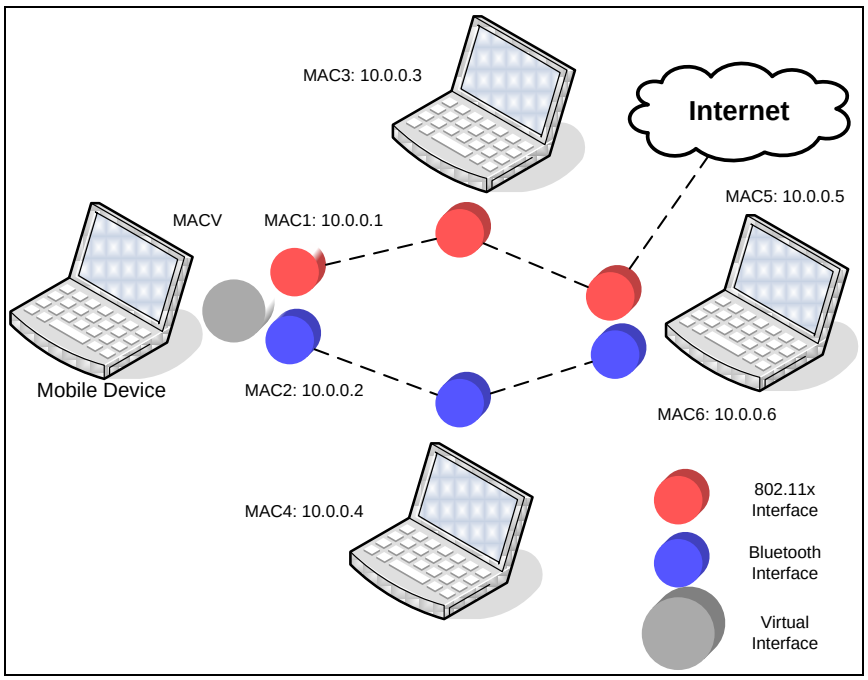


Figure 1.1: Ad-Hoc Scenario A with several access technologies (Bluetooth and 802.11x).

The scenario presented in Figure 1.1 is a particular but good example of the heterogeneity presented in wireless networks. Albeit with capability to be used simultaneously, the current drivers of the mobile device do not take this into consideration. Our proposal is to consider a transparent way to make full use of all the interfaces by developing an abstraction interface, i.e. a *virtual aggregation interface*. This virtual interface is responsible for monitoring the physical network interfaces, and based on their availability as well as on our implemented policies mechanisms, it will use one or several simultaneously. Expectations are that it may increase throughput, improve energy efficiency, as well as potentially reduce the network latency.

In this type of scenario, some questions we shall consider and attempt to answer are:

- Is it possible to consider such a global virtual interface both from a network and from a user perspective?
- If technically this is feasible in a way that optimizes network efficiency, what are the technical implications of developing such interface?
- Intuitively, in terms of network robustness there are clear benefits to consider, but there are also open issues which need to be analyzed, e.g., RTT delay and reordering consequences.

A second applicability scenario, our proposal addresses, relates to multihomed mobile devices that interconnect to different access networks. This is often the case today for residential users that have e.g. at least one fixed line connection terminated by Wi-Fi, and a 3G

connection. By developing adequate virtualization, one can assist in the optimization of mobility (vertical handover scenarios) from a user and network perspective. See Figure 1.2 for a conceptual representation, where we have different access networks (3G and Wi-Fi), and a virtual interface, is deciding which interface(s) to use in a certain moment.

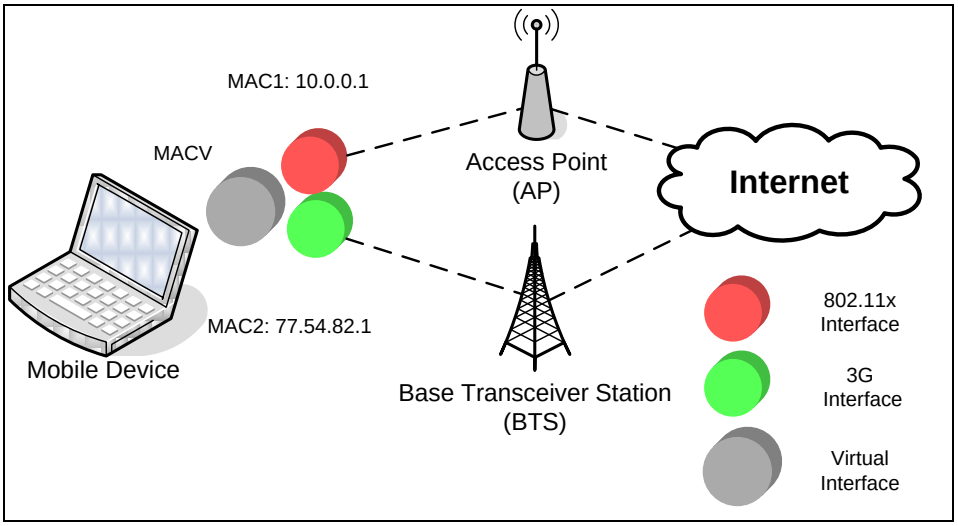


Figure 1.2: Scenario B with several access technologies (3G and 802.11x).

1.2 Goals, Assumptions, and Expected Results

The main goals of this work are:

- To conceive and to develop cooperative access mechanisms which assist in distributing information to several users based on virtualization techniques that assist multiple network interfaces to be transformed into a single interface, for both, the network and the applications.
- To improve the efficiency of heterogeneous wireless networks by considering dynamic and intelligent load-balancing techniques across different available and active interfaces, with and without virtualization.
- To optimize horizontal handovers and Quality of Service based on the developed mechanisms.
- To understand the impact of the developed solutions on current Internet wholesale models.

Our goal is to integrate heterogeneous mobile ad-hoc networks that use different wireless network technologies and conceive/develop cooperative access mechanisms which assist in

distributing information to several users. Although we are interested in a generic solution, we take a network that combines different flavors of 802.11x as a basis for this thesis.

This thesis is focused on a promising end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc networks. The solution is based on a *virtual interface* (vi) approach, which allows the usage of all interfaces presented in a mobile device simultaneously, while hiding the heterogeneity from the network and allowing any number of interfaces to be added, increasing the total throughput.

The contributions of the thesis can be enumerated as follows:

- A brief survey, analysis and comparison of previous work done in this area.
- An end-to-end communication abstraction, also known as virtual interface.
- A method of intercepting the data and relaying it to the virtual interface without adding excessive overhead.
- Several mechanisms which will allow a throughput gain, by exploring the simultaneous usage of several physical interfaces present in a mobile device.
- Power saving mode, choosing the interfaces which will transfer a certain data flow while consuming the lowest amount of energy.

2 State-of-the-Art

This work addresses the efficient utilization of multiple network interfaces of a single device by devices, via rate control and optimal assignment of traffic flows to available networks, with a special emphasis on Ad-Hoc networks. This section introduces fundamental concepts, starting with a brief overview of the wireless architectures, followed by an analysis of current related research.

Section 2.1 gives an overview of some short-ranged wireless technologies, such as *IEEE 802.11* (Wi-Fi) and *IEEE 802.15.x* (Bluetooth) and introduces the evolution of the wireless networks, starting with the infrastructure mode, followed by the ad-ahoc mode.

Section 2.2 reviews some related technologies, such as Femtocells and Smart APs, used to empower the end-user.

Section 2.3 describes how to deal with multiple interface devices, with a special emphasis to the multihoming and load balancing mechanisms, features, drawbacks (such as handover), advantages, as well as the interface virtualization technique, given that one of the main goals of this thesis is to improve the cost-effective connectivity, which will require the virtualization of interfaces, to handle different physical interfaces in a transparent way.

Section 2.4 reviews some Linux networking components, how they work and interact with each other. In this section, some of these components, such as *Netfilter* and *iptables* are presented. Finally, in section 2.5 there is a small discussion identifying the problems and drawbacks in the surveyed work.

2.1 Emerging Wireless Architectures and Technologies

The explosive growth of the Internet over the last decade has led to an increasing demand for high-speed, ubiquitous Internet access. Broadband Wireless technologies are increasingly gaining popularity by the successful global deployment of the *Wireless Personal Area Networks* (Bluetooth- IEEE 802.15.1), *Wireless Local Area Networks* (WiFi- IEEE 802.11x), and *Wireless Metropolitan Area Networks* (WiMAX-IEEE 802.16) [1]. Using open broadband Wireless technologies and implementing mobile computing architectures, one can overcome the challenges of ground, infrastructure, and finance to increase access; deploy broadband quickly and cost-effectively to areas currently not served; and extend the benefits of digital revolution to previously unreachable populations.

In this section we will only be introducing Wireless Personal Area Networks (Bluetooth-IEEE 802.15.1) and Wireless Local Area Networks (WiFi-IEEE 802.11x), as these are the technologies that fall under the scope of this thesis.

2.1.1 Short-range Wireless Technologies

Emerging technologies such as *Bluetooth (BT)* and 802.11b (Wi-Fi) have fuelled the growth of short-range communication industry. The differences between their standard features (data rate, distance range, security, and communication protocol) have lead to a natural partitioning of applications.

2.1.1.1 Bluetooth and Wi-Fi

BT, the leading WPAN technology, was designed primarily for low-cost cable replacement. On the other hand Wi-Fi, today the most popular short-range wireless technology, was initially conceived as a simple and plug&play way to extend the reach of fixed lines – Wi-Fi is based on the Ethernet standards. Nonetheless, the fact is that today most UEs for personal use such as laptops and PDAs, require both BT and Wi-Fi standards to cover a wider range of applications in both the home and office spaces.

Wireless communication systems use one or more carrier frequencies (frequency bands) to communicate. Bluetooth and Wi-Fi share the same 2.4 GHz band, which under *Federal Communications Commission* (FCC) regulations, extends from 2.4 to 2.4835 GHz. Under the ISM band rules defined in FCC Part 15.247, this frequency band is free of tariffs. It is license exempt in Europe. However, systems must operate under certain constraints that are supposed to enable multiple systems to coexist in time and place.

These two technologies are described in more detail in the annex section, since they are important but not essential to the understanding of the proposed solution.

2.1.2 Wi-Fi Modes of Operation

There are two different models for Wi-Fi networks that exist today: *Infrastructure* mode and *Ad-Hoc* mode.

In Infrastructure mode the wireless network consists of at least one access point connected to the wired network infrastructure and a set of wireless end stations. This configuration is called

a *Basic Service Set* (BSS). An *Extended Service Set* (ESS) is a set of two or more BSSs forming a single subnetwork. Since most corporate WLANs require access to the wired LAN for services (file servers, printers, and Internet links) they will operate in infrastructure mode (cf. Figure 2.1).

A big advantage of this model is the possibility for the network (and consequently the access operator) to better manage resources. The flip-side is that it requires some specific hardware and some previous planning of the network, undermining the possibility for users to start communication sessions spontaneously, where and whenever they want.

Ad-Hoc mode (also called peer-to-peer mode or an *Independent Basic Service Set*, or IBSS) is simply a set of 802.11 wireless stations that communicate directly with one another without using an access point or any connection to a wired network. This mode is useful for quickly and easily setting up a wireless network anywhere that a wireless infrastructure does not exist or is not required for services [3], such as a hotel room, convention center, or airport, or where access to the wired network is blocked (cf. Figure 2.1).

The routing in mobile ad-hoc networks necessitates specialized algorithms and protocols that can cope with the dynamic nature of appearing and vanishing neighbors. Two major protocols have been used in this work, not only to test the virtual interface in a realistic environment, but also to update the information regarding the available neighbors of the mobile device. A more indebt description of the two algorithms can be found in the annex section of this thesis (cf. Annex II).

Today, the most common instances of ad-hoc networks are *Mobile Ad-Hoc Networks* (MANETs), and mesh networks. A MANET is purely an ad-hoc network where some nodes move. While a mesh network is considered to be a set of nodes (multihop or not) which would be static [2].

A more recent type of wireless architecture is a *user-provided network* (UPN). A UPN is a wireless network (be it infrastructure, ad-hoc, or mesh) [4], which is triggered by the willingness of some end-users to cooperate (based on cooperation incentives) in a spontaneous way. UPNs are architectures that operate in isolation or as complement to access technologies (e.g. 3G) being the main difference to the older wireless architectures the fact that networking nodes are controlled partially by the end-user. It should be noticed that the aspect of having some access control moved to the end-user is an essential aspect that is being pursued also from an access perspective, as can be seen in section 2.2, where femtocells and smart APs are addressed.

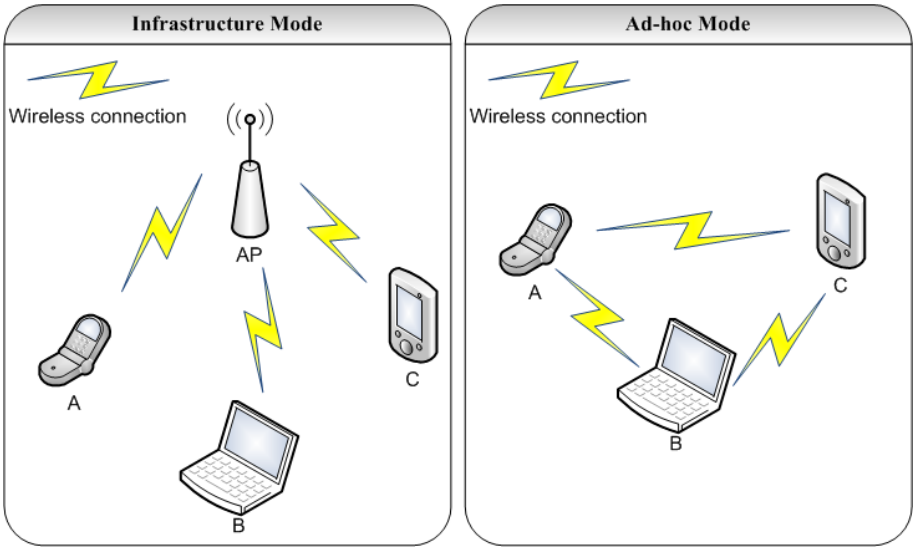


Figure 2.1: infrastructure and ad-hoc modes.

2.2 Empowering the end-user: Femtocells and Smart APs

In this section some technologies, such as Femtocells and smart APs, responsible for empowering the end-user connectivity are presented. These kinds of technologies, in some circumstances, improve the network signal by automatically taking certain decisions for the user, making use of their knowledge about the available network resources.

2.2.1 Femtocells

Femtocell is a recent technology which uses the IP backbone network along with small-size base stations, based on cellular technology, located indoors. Doing so, femtocells support compatibility with the cellular systems, and at the same time, provide better indoor signal strength [10], commonly unattainable by macrocell coverage operating at higher frequencies.

The femtocell appears to the standard 3G phone as just another cell site from the host mobile operator, and can be used by almost any 3G phone including roamers visiting from other countries.

The mobile operators telephone switch (MSC) and data switch (SGSN) also communicate to the femtocell gateway in the same way as for other mobile calls. Therefore, all services including phone numbers, call diversion, voicemail etc. all operate in exactly the same way and appear the same to the end user.

The connection between the femtocell and the femtocell controller uses secure IP encryption (IPsec), which avoids interception and there is also authentication of the femtocell itself to ensure it is a valid access point. Figure 2.2 illustrates the system architecture and context for femtocell operation.

Inside the femtocell are the complete workings of a mobile phone *base station* (BTS). Additional functions are also included such as some of the *Radio Network Controller* (RNC) processing, which would normally reside at the mobile switching centre. Some femtocells also include core network element so that data sessions can be managed locally without needing to flow back through the operators switching centers.

To summarize, the capacity benefits of femtocells are attributed to:

- Reduced distance between the femtocell and the user, which leads to higher received signal strength;
- Lowered transmit power, and mitigation of interference from neighboring macrocell and femtocell users due to outdoor propagation and penetration losses;
- As femtocells serve only around one to four users, they can devote a larger portion of their resources (transmit power and bandwidth) to each subscriber. A macrocell, on the other hand, has a larger coverage area (500 m–1 km radius) and a larger number of users; providing *quality of service* (QoS) for data users is more difficult.

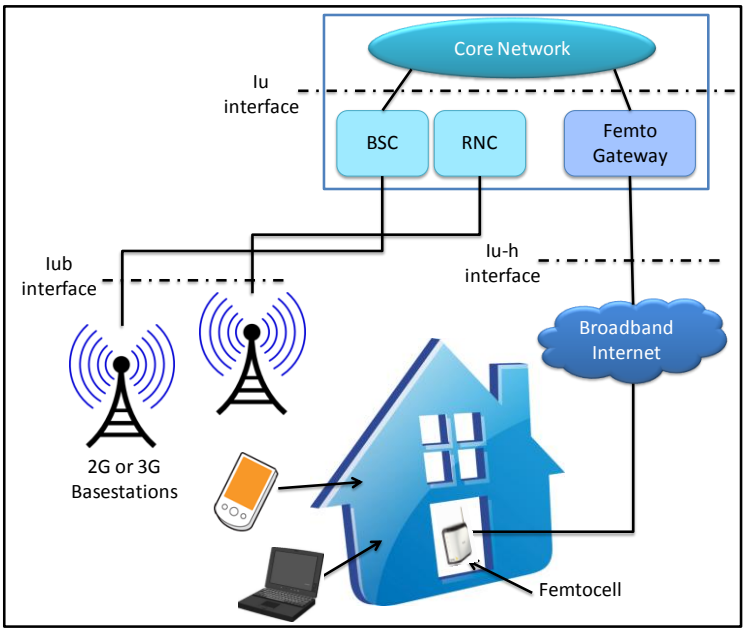


Figure 2.2: System architecture and context for femtocell operation.

2.2.2 Smart APs

Wireless access points are rapidly increasing in number and variety, providing people with connectivity in almost all buildings they enter (e.g. home, work place, etc.). The wireless medium, in fact, is naturally prone to be shared by several users who may interfere with each other, harming the performance of UDP-based real-time flows (e.g., online gaming) as TCP continuously probes the channel for more bandwidth, thus eventually generating queues (delays) on the connection [11].

A smart Access Point can take advantage of its knowledge about available wireless network resources and the on-going traffic in order to appropriately limit TCP's advertised windows so as to smooth the network traffic progression and avoid queuing delays [12]. Furthermore, a smart Access Point also provides radio functionality and has most of its network intelligence in the same box, thus these devices can handle most of the protocols for roaming, encryption, management, user authentication, and so forth. A smart AP presents the end-users it serves to the wired network switch as if they were physically connected, reducing the load on central switches within the wired LAN, albeit at the cost of needing to be managed [13].

Integrating network services directly into the AP also enables important services to be pushed out to the first point of contact with the wireless user. The thought is that by provisioning access control lists and policies directly from the radio function, end-users can move, for example, onto another subnet in another corporate location, and still retain all their access rights.

2.3 Dealing with Multiple Interfaces

Today's end-user devices are equipped with several network interfaces and have at their disposal a multitude of applications with different bandwidth requirements. To make use of this variety, some mechanisms such as multihoming, load-balancing, bandwidth aggregation and interface virtualization, have been used to grant end-user with some redundancy, help solving some mobility problems and make a better use of all available interfaces. In this section we introduce some of these solutions and survey some work done in this area.

2.3.1 Multihoming

In multihoming, a single computer host makes use of several IP addresses associated with various connected networks. Within this scenario, the multihomed computer host is physically linked to a variety of data connections or ports. These connections or ports may all be associated with the same network or with a variety of different networks. Depending on the exact configuration, multihoming may allow a computer host to function as an IP router.

One possibility for the process of multihoming makes use of what is known as *Stream Control Transmission Protocol*, or SCTP. Essentially, the process involves employing multihoming by making use of a single SCTP endpoint to support the connectivity to more than one IP address. By establishing connection to multiple addresses, multihoming can help to enhance the overall stability of the connectivity associated with the host [14].

One of the advantages of multihoming is that the computer host is somewhat protected from the occurrence of a network failure. With systems that make use of a single IP address and connection, the failure of the connected network means that the connection shuts down, rendering the end system ineffectual as far as connectivity to the Internet is concerned. With multihoming, the failure of a single network only closes a single open door. All the other doors, or IP addresses associated with the other networks, remain up and functional.

Multi-homed networks are often connected to several different *Internet Service Providers* (ISPs). Routers use *Border Gateway Protocol* (BGP), a part of the TCP/IP protocol suite, to route between networks using different protocols [15].

In general, multihoming is helpful for three elements of effective web management. First, multihoming can help to distribute the load balance of data transmissions received and sent by the computer host by lowering the number of computers connecting to the Internet through any single connection. Second, the redundancy that is inherent to multihoming means less incidences of downtime due to network failure. Last, multihoming provides an additional tool to keep network connectivity alive and well in the event of natural disasters or other events that would normally render a host inoperative for an extended period of time [14].

2.3.2 Load-balancing

In computer networking, load-balancing is a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload. Using multiple components with load-balancing, instead of a single component, may increase

reliability through redundancy. The load-balancing service is usually provided by a dedicated program or hardware device (such as a multilayer switch or a DNS server).

One of the most common applications of load-balancing is to provide a single Internet service from multiple servers, sometimes known as a server farm.

A variety of scheduling algorithms are used by load balancers to determine which backend server to send a request to [16]. Simple algorithms include random choice or round robin. More sophisticated load balancers may take into account additional factors, such as a server's reported load, recent response times, up/down status (determined by a monitoring poll of some kind), number of active connections, geographic location, capabilities, or how much traffic it has recently been assigned. High-performance systems may use multiple layers of load-balancing.

2.3.3 Network Switching

The authors of *On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts* study if heterogeneous wireless interfaces can be aggregated with intelligent strategies to improve throughput beyond sum of the parts, as they call them *super-aggregation* principles. The authors propose three principles in the context of TCP that achieve *super-aggregation* benefits in Wi-Fi network when by adding a 3G interface [21]:

- S*elective offloading*: some of the interfaces may have a limited bandwidth, and by selectively offloading some portions of the data transferred it can cause a significant impact on the performance.
- *Proxying*: when an interface has only limited bandwidth but is up when the other interface is down, the limited bandwidth can be used for critical control information that in turn can serve to significantly improve the overall performance of the data transfer.
- *Mirroring:* for certain portions of the data being transferred intelligently mirroring the transfer on the interface with lower bandwidth can again have a profound impact on the perceived performance.

The *super-aggregation* principles presented can be implemented as a layer-3.5 software middleware in the mobile host. It can be implemented in the Linux kernel and uses NetFilter [22] to capture and process TCP packets traversing the network stack, or generate packets if necessary. The *super-aggregation* principles only require deployment at the mobile device and do not require any modification at the remote host or intermediate routers. The TCP implementations on the remote host and the mobile device are unaware of the *super-aggregation* principles that improve their performances transparently [21]. With this deployment

model, *super-aggregation* can enhance end-to-end performance of mobile host with any legacy TCP-based server.

This solution although making possible the usage of two interfaces simultaneously (in this case Wi-Fi and 3G) and increasing the total throughput, does not escalate to more interfaces, does not take in consideration the use of two interfaces with similar bandwidth since it uses the interface with lower transmission rate to send certain small messages (e.g. ACK messages) and the other interface to send and receive the remaining data.

The tests prove that their solution in fact provides clear improvements in terms of throughput beyond the sum of the parts, which did not happen with other simple aggregation solutions [23][24][25], but unfortunately the authors only tested their solution with TCP data, neglecting the UDP data.

2.3.4 ISP Switching

The authors of [17] investigate the feasibility of switching among ISPs to exploit the benefits of choosing the ISP with better connectivity conditions (in the sense of a cost function that may include throughput, access cost, among others) at any given time. This approach is suitable for connections with long duration such as file transfer or streaming applications that need to switch from a congested ISP to a non-congested one, if available.

To analyze topological path diversity, the authors use *traceroute* data. They first trace the end-to-end paths from the test-bed to each of the destinations through both ISPs. To resolve IP aliases the authors use *sr-ally* [17].

To measure latency, loss and jitter on both ISPs, the authors send probe packets simultaneously through two network interfaces so that the probe packets travel through both ISPs at the same time. To measure the round-trip time (latency) and packet loss ratio they use *ping*. Since it is desirable to have no (or low) overlap among the alternative paths provided by multihoming, the authors also defined the metric *Single Source Path Overlap* (SSPO) to express the path overlap between a multihomed user and any host in the network. As shown in Figure 2.3, the path overlap occurs for a multihomed host at the edge network with which the source node is connected. SSPO is an estimation of the expected fraction of hop overlap, which is the ratio of the shared hops to the total non-shared hops of all paths.

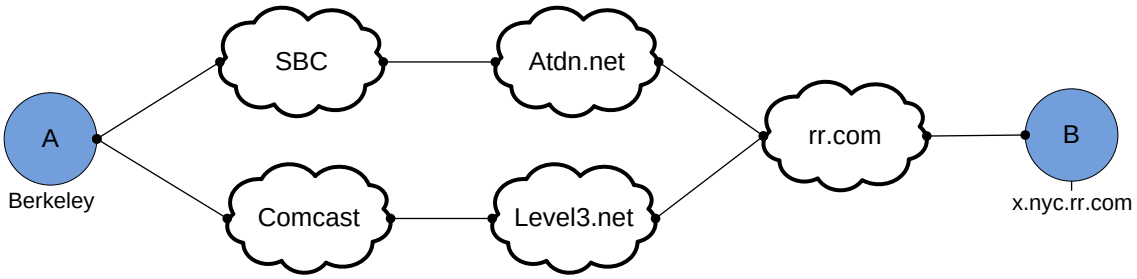


Figure 2.3: Path diversity, user A has two separate paths to reach user B.

The key insight into the potential benefits of multihoming is that not only it provides first hop path redundancy, but more generally it offers highly diverse end-to-end paths both in topology and network layer metrics such as latency, loss, and jitter [18].

2.3.5 Aggregation: Interface virtualization

In this section, we introduce the interface virtualization technique, which intends to hide the heterogeneity created by the use of all network interfaces, presented in mobile devices, from the applications, with special emphasis to one specific work, presented in section 2.4.5.3 that implements a virtual interface as a layer two device, capable of hiding all physical interfaces under it, which ended up being a base for this thesis.

2.3.5.1 Interface Virtualization

Virtualization was first introduced in the 70's by IBM as a way to assist in supporting concurrent processes on a single machine. Out of these emerged different systems which are today common in any machine.

Within the context of wireless networks, virtualization is being heavily applied due to the rise of *Software Defined Radio* (SDR). SDR gives the means not only to take better advantage of a single wireless interface (multiplexing) but also to consider aggregating radio resources in a way that makes the network more robust. But the key aspect in virtualization applied to wireless networks is that such a system can take advantage of a multitude of proprietary radio technologies in a way that makes it transparent to the application and to the end-user. A technology that is often used with SDR is the *Cognitive Radio* (CR), a form of wireless communication in which a transceiver can intelligently detect which communication channels

are in use and which are not, and instantly move into vacant channels while avoiding occupied ones. This optimizes the use of available *radio-frequency* (RF) spectrum while minimizing interference to other users.

On the other hand, a software radio *Base Transceiver Station* (BTS) is much more readily virtualized than a hardware radio, since the BTS is just a software application. It is possible to construct a virtualized base station by using standard virtualization technology to create a virtual machine (VM) per operator, and running an independent BTS application for each operator within that VM. This ensures that each operator has complete control over their BTS, while guaranteeing that one operator's traffic, signaling and configuration data are isolated from other operators.

A *virtual interface* approach is the most convenient solution for heterogeneous mobile ad-hoc networks in terms of transparency [20]. To provide a faster path between applications and the network, most researchers have advocated removing the operating system kernel and its centralized networking stack from the critical path and creating a *userlevel network interface* [19]. With these interfaces, designers can tailor the communication layers each process uses to the demands of that process. Consequently, applications can send and receive network packets without operating system intervention, which greatly decreases communication latency and increases network throughput [20].

Figure 2.4 shows system memory with two applications accessing the network through a user-level network interface. A device driver in the operating system controls the interface hardware in a traditional manner and manages the application’s access to it.

Applications allocate message buffers in their address space and call on the device driver to set up their access to the network interface. Once set up, they can initiate transmission and reception and the interface can transfer data to and from the application buffers directly using direct memory access.

User-level network interface designs vary in the interface between the application and the network. How the application specifies the location of messages to be sent, where free buffers for reception get allocated, and how the interface notifies the application that a message has arrived. Some network interfaces, such as Active Messages or Fast Messages, provide send and receive operations as function calls into a user-level library loaded into each process.

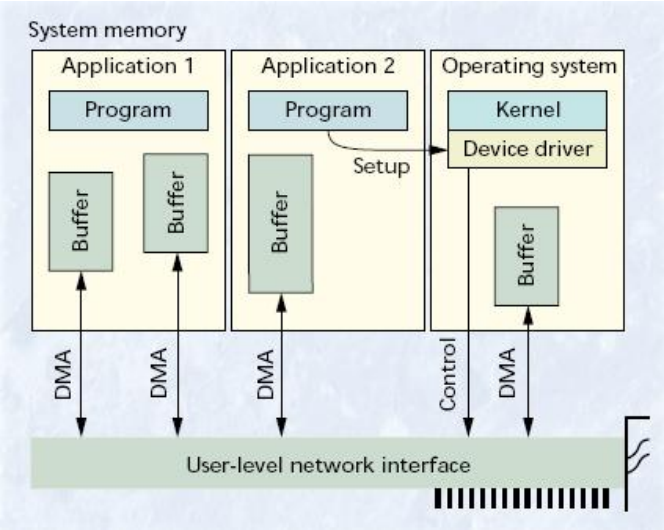


Figure 2.4: System memory using a user-level network interface. Figure extracted from [19].

2.3.5.2 Bluetooth Network Encapsulation Protocol – BNEP

The *Personal Area Networking* (PAN) Bluetooth Network Encapsulation Specification describes the protocol to be used by the Bluetooth PAN profiles. The document [26] defines a packet format for Bluetooth network encapsulation used to transport common networking protocols over the Bluetooth media. Bluetooth network encapsulation supports the same networking protocols that are supported by IEEE 802.3/Ethernet encapsulation. Packets from the supported networking protocols are contained in Bluetooth network encapsulation packets, which are transported directly over the Bluetooth L2CAP protocol.

In the scope of this thesis, BNEP can be used to encapsulate Bluetooth packets, making it possible to create an ad-hoc network that contains devices with both Bluetooth and 802.11x physical interfaces, which makes it possible to hide both type of interfaces under one virtual interface.

2.3.5.3 Transparent Heterogeneous Mobile Ad-Hoc Networks

The authors' of this work [28] goal was to develop an end-to-end communication abstraction that supports MAC-switching[1], node mobility and multihoming[1]. Two issues to be solved are

[1] Refers to the fact that the used MAC technology may change along a source/destination path.

broadcast emulation and handover. Broadcast emulation because broadcast is not directly supported in Bluetooth (or on nodes comprising both Bluetooth and 802.11).

Handover is an issue because, in the case of heterogeneous mobile ad-hoc networks, a handover might include a change in how the medium is accessed. A handover can be caused by node mobility, a change in user preferences (the user chooses to save energy and use Bluetooth instead of 802.11), or performance reasons.

The proposed solution is inspired by the Linux Ethernet Bridge [27]. Similar to a physical bridge device, the Linux Ethernet bridge ties separate layer-two-networks together, only that it is purely software. It appears to the operating system as a regular layer-two-device one can easily assign IP addresses to bridges. The bridge is supposed to work in combination with 802.x devices, therefore not including Bluetooth. Fortunately, the Bluetooth personal area network (PAN) profile specifies BNEP [26] which itself defines a packet format to transport common networking protocols over the Bluetooth media. BNEP supports the same networking protocols that are supported by IEEE 802.3/Ethernet encapsulation, therefore enabling Bluetooth to be used within the bridge.

The authors define a Virtual Interface (*vi*) that is responsible for storing a MAC/Interface mapping, based on incoming packets. Like a Linux ethernet bridge, the *vi* represents a regular layer-two-device and can be configured accordingly. The *vi* allows to plug in any 802.x compatible network device, like e.g a wireless LAN card or a BNEP/Bluetooth connection, while hiding the heterogeneity of the used devices from the upper layers. For every neighbouring node, the *vi* holds an array of possible outgoing interfaces in a so called neighbouring database. The author's solution is not bound to 802.11x or Bluetooth, but works together with any 802.x-compatible MAC Layer. The *vi* in combination with a MANET routing protocol supports multihoming, dynamic reconfiguration and node mobility.

If the *vi* receives a packet from the upper layer for delivery, it first checks the packet type. In case the packet is a broadcast packet, it will be sent through all available interfaces. Therefore, the *vi* also acts as a broadcast emulation layer for Bluetooth. However, if the packet is unicast, the *vi* looks for the corresponding entry in the neighborhood database mentioned above and retrieves the information about the interface the packet has to be sent to (entries are periodically checked for expiration). If there is more than one option, the *vi* makes use of another feature, the so called priority table. The priority table specifies a ranking among the interfaces, meaning that whenever a given neighbour can be reached through several interfaces, the interface with the lowest priority is taken. This means that the *vi* also acts as a load-balancing mechanism, capable of prioritizing interfaces based on different factors (e.g. energy consumption).

---

[1] A node having multiple network interfaces.

The authors also introduce a parameter that is associated with a *vi*, the so called *maxdiff* threshold. The *maxdiff* threshold unit is 10ms and it decides how much two single entries within the neighborhood database may differ in terms of timestamps to keep the priority policy up. So a higher ranked interface entry can be replaced by a lower priority interface if the timestamp differs for more than *maxdiff* [28].

The experiments presented demonstrate the feasibility of the abstraction and its potential in building heterogeneous ad-hoc wireless networks. The measurements show that the system performs well. There is almost no overhead when using the additional *vi* on top of a physical interface. Generally, the *vi* performs better in combination with AODV than with OLSR. Except for the case of priority driven handover, the *vi* also works without a MANET routing module. However, a routing module increases the performance in terms of packet loss during handover. In the case of multiple physical interfaces there is a trade-off between agility in terms of vertical handover and throughput for UDP traffic [28]. The evaluation is clear for the same technology, but not when relying on different technologies.

Even though this work presents an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc network, it does not make full use of the interfaces presented in mobile devices. Meaning, this solution does not offer the possibility to use both interfaces simultaneously, to send different traffic flows of information in order to increase the overall transmission rate.

2.3.5.4 Linux Ethernet Bridge

The Linux Ethernet Bridge allows putting several real interfaces into a virtual bridging device [29]. It is not only an in-kernel equivalent to a real Ethernet bridge but together with *Netfilter* a very sophisticated tool for packet filtering. Packets are forwarded based on Ethernet address, rather than IP address (like a router). Since forwarding is done at Layer 2, all protocols can go transparently through a bridge. The Linux bridge code implements a subset of the ANSI/IEEE 802.1d standard.

Bridging is supported in the 2.4 and 2.6 kernels from all the major distributors. The required administration utilities are in the *bridge-utils* [29] package in most distributions.

An Ethernet bridge distributes Ethernet frames coming in on one port to other ports associated to the bridge interface. Whenever the bridge knows on which port the MAC address to which the frame is to be delivered is located, it forwards this frame only to this port instead of polluting all ports together. Ethernet interfaces can be added to an existing bridge interface and become then (logical) ports of the bridge interface.

Putting a *Netfilter* structure on top of a bridge interface renders the bridge capable of servicing filtering mechanisms. This way, a transparent filtering instance can be created. It even needs no IP address assigned to work. Of course, an IP address can be assigned to the bridge interface for maintenance purposes.

The advantage of this system is evident. Transparency alleviates the network administrator of the pain of restructuring the network topology.

2.4 Linux Kernel Aspects

To fully understand this thesis and all tools used in its implementation, it is essential to understand how some Linux networking components work, and how they interact with each other. In this section, some of these components, such as *Netfilter* and *iptables* are presented.

2.4.1 Netfilter

Netfilter [31] consists of a number of hooks at various points inside the Linux protocol stack. It allows user-defined kernel modules to register callback functions to these hooks. When a packet traverses a hook, the packet flows through the user defined callback method inside the kernel module. The Linux kernel contains many so-called *Netfilter* targets to build powerful packet filter rule sets. *Netfilter* can intercept packets at many states of their processing and perform arbitrary operations on them.

The *Netfilter* framework has been incorporated into the Linux kernel 2.4 or later versions to replace the old ipchains architecture [31]. In the new architecture, the *iptables* command, which will be shortly described in the next section, is a user-space program that can configure kernel-space modules such as firewall, network address translation, port-forwarding, and QoS. All filtering rules, including the proposed improvement, can be added into the kernel-space via the *iptables* command.

The *Netfilter* framework has five hooking points in the kernel as shown in Figure 2.5:

- NF_IP_PRE_ROUTING,
- NF_IP_LOCAL_IN,
- NF_IP_FORWARD,
- NF_IP_LOCAL_OUT, and

- NF_IP_POST_ROUTING.

NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING are the hooking points for packets that enter and leave the system, respectively. NF_IP_LOCAL_IN is the hooking point for packets that are redirected to the user-space local processes after entering the system.

NF_IP_LOCAL_OUT is the hooking point that packets leave the user-space local processes. NF_IP_FORWARD is the hooking point that packets are forwarded from one network interface to another.

Packets will pass through hooking points sequentially. On each hooking point, users may configure some filtering rules via the *iptables* command. After packets pass through NF_IP_PRE_ROUTING, the Linux kernel makes the routing decision to decide whether packets should enter the local processes or be routed to the next hop through another network interface. Hook functions will return one of five kinds of results when a packet passes through each hooking point.

The five possible results include:

- NF_DROP: to drop the packet;
- NF_ACCEPT: to accept and forward the packet to the next hooking point or network interface;
- NF_STOLEN: temporarily to ignore the packet and process the packet later, like modifying the contents of the packet;
- NF_QUEUE: to store the packet that will be later examined by other user-space processes, like snort;
- NF_REPEAT: to return the packet to the current hooking point in order to match other rules.

A basic firewall requires only NF_DROP and NF_ACCEPT states to satisfy users' needs.

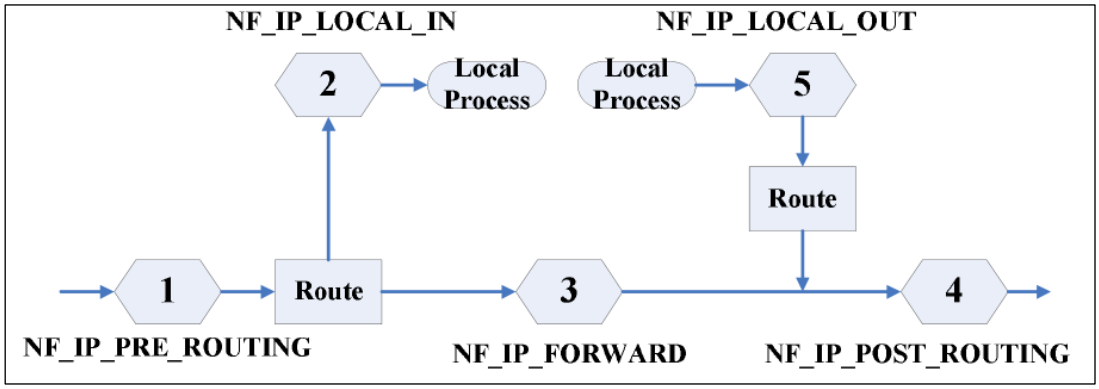


Figure 2.5: Hooking points in Netfilter.

In the scope of this thesis, *Netfilter* can be used to intercept the data coming from and to the application, redirecting it to the virtual interface, which will then decide what to do with that information.

2.4.2 Iptables

"*iptables*" is a user-space program for users to configure filtering rules or network address translation rules into Linux *Netfilter* kernel modules. There are three kinds of tables that *iptables* can configure. They are filter, nat, and mangle tables. Each table is associated with one of the major functions of *Netfilter* [35]. Users can configure *iptables* rules based on the three tables/functions. Because *iptables* rules set in each table may be activated in different hook points, rules of the same table may then be partitioned into different chains each of which corresponds to a hook point. Table 2.1 shows the relationship between tables and chains.

Table	Chain
Filter	INPUT FORWARD OUTPUT
Nat	PREROUTING OUTPUT POSTROUTING
Mangle	PREROUTING INPUT FORWARD OUTPUT POSTROUTING

Table 2.1: Chains used in each table.

Table 2.2 shows an example of an *iptables* rule. The “-t” option describes which table the rule should be set into. The “-A” option describes which chain the rule should be added in. This option also indicates the corresponding hook point on which the rule may be applied. The “-p” option indicates which type of data we are referring to. The “-i" option describes which physical interface the rule is being applied to. The “--dport" describes the port, in this case it is the port assigned to the ssh protocol, which by default is port 22. The last “-j ACCEPT” indicates that the packet should be accepted when matching the rule.

iptables –t mangle -A INPUT -p tcp -i eth0 --dport ssh -j ACCEPT

Table 2.2: iptables rule example.

A more complete and advanced look into the kernel world as well as introduction to a few of the basic concepts of Linux kernel programming is given in Annex III of this thesis.

2.5 Discussion

As seen in the above sections there is some research done regarding mobility management and multihoming; however, there is not an adequate solution, that makes full and simultaneous usage of all interfaces present in a mobile device, not only in terms of adequate load-balancing, but also in considering the physical aspects of the various interfaces/channels to send different traffic flows of information to one or more networks.

The paper presented by the authors of the *Transparent Heterogeneous Mobile Ad-Hoc Networks* (Section 2.3.5.3) is the closest work we found to our proposed solution. We implemented a similar solution, using a virtual interface to hide the heterogeneity of the used devices from the upper layers, improve the way the priorities table is set and add the possibility to use several different flavors of Wi-Fi interfaces, in order to increment the total throughput, via an aggregation method similar to the one presented by the author of the paper *On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts* (Section 2.3.3).

Also, it is relevant to highlight that in all the surveyed work, the authors only tested their solutions performance in terms of throughput and did not assess the energy being consumed. Together with our load-balancing mechanism we have also implemented a power consumption algorithm that measures the amount the energy that is being consumed, and determines which network interfaces should be used to save the most amount of energy, when the device is running on low battery levels.

3 Our Proposed Architecture

Nowadays, mobile devices are equipped with several physical interfaces, together with some multihoming techniques and load-balancing mechanisms, allowing them to change the interface being used according to its availability. As explained in the previous section, there is no solution available that provides the user equipment both with the capability to perform multihoming and with the capability to use all available interfaces simultaneously to send different traffic flows in a balanced and transparent way.

We implemented a virtual interface that is able to perform load-balancing and also analyze each equipment requirements using a priority and a neighboring database table. This virtual interface (*vi*) besides hiding the heterogeneity from the application, aggregates the physical interfaces under it in a transparent way, decides which interfaces should be used and if needed, will perform the handover in case an interface is no longer available. The architecture of the implemented *vi* will be explained in more detail during the next sections.

Although we are interested in a generic solution, we take a network that combines different flavors of 802.11 as a basis for this thesis, in particular to assist realistic experimentation.

This section starts by providing an informal description of the environment being targeted. Then, a more detailed model for this environment will be presented, capturing the assumptions made about the network architecture limitations. Finally, the implemented architecture and approach to take, that transparently improves the cost-effective connectivity in heterogeneous access networks, will be provided.

3.1 Architecture Model

In terms of architecture, it is divided into 4 main blocks:

- Virtual Interface;
- Priority Table;
- Decider / Virtual Bandwidth Aggregation (VBA);
- RTT Estimator.

These blocks will work together under a single kernel module to insure the correct distribution of data trough the several existing physical interfaces [33]. Figure 3.1 describes the path that the data coming from a certain application takes, until it reaches the physical interfaces, passing through our virtual interface.

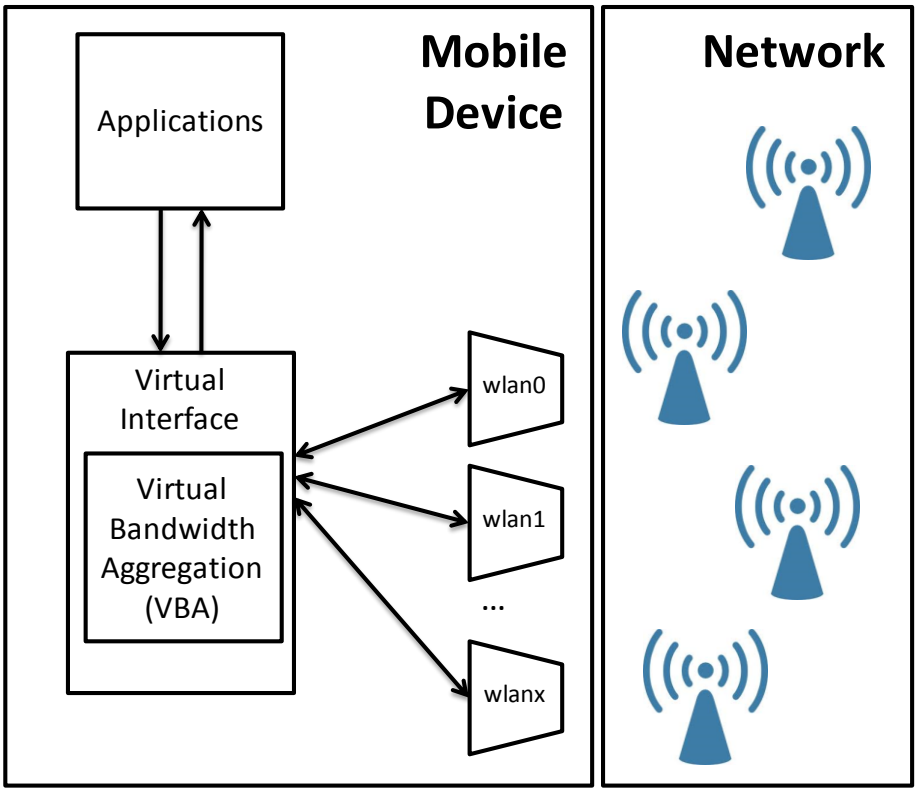


Figure 3.1: Interaction between the implemented mechanisms.

The data coming from and to the application is intercepted by the virtual interface, which will check three parameters: the priority, availability and RTT of each interface. Then the *Virtual Bandwidth Aggregation* block (VBA) (cf. section 3.1.1) will decide how to distribute the intercepted data between the available physical interfaces (cf. Figure 3.1).

The VBA and RTT Estimator were created from scratch in order to choose which mechanism to use based on the available interfaces, the other two blocks were modified and updated in order to properly function with the most recent kernel versions and to correct some problems related with the significant overhead that was being added by the hooks placed by the previous authors [28][32].

3.1.1 Virtual Interface

Figure 3.2 illustrates the architecture of the virtual interface (*vi*) and how it is embedded within the network stack. We have the *vi* hiding the heterogeneity from the application, connected to several physical interfaces. As seen in the figure, the *vi* will also communicate with the virtual bandwidth aggregation block (VBA) which is described afterwards.

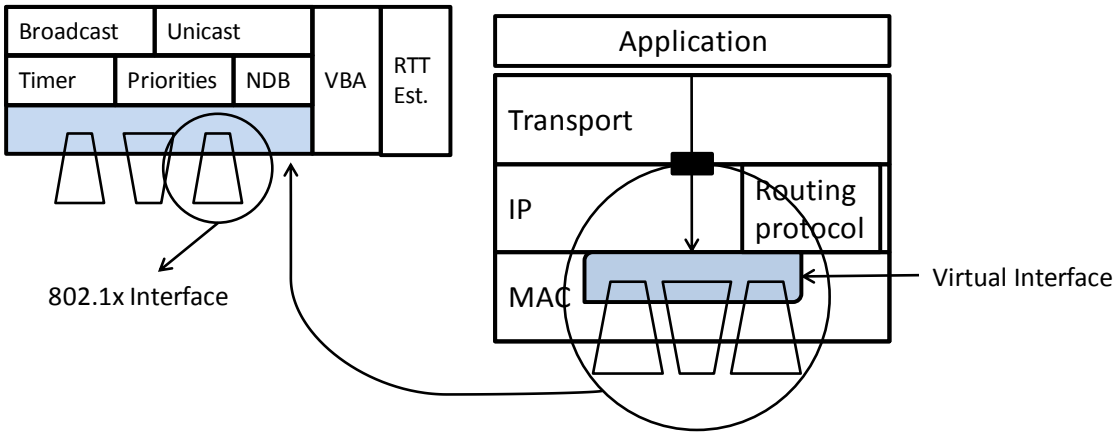


Figure 3.2: Virtual interface architecture approach.

The virtual interface is similar to the Linux Ethernet Bridge [33]. The *vi* represents a regular layer-two-device and can be configured accordingly and supports any 802.x compatible network device, such as a wireless LAN card or a BNEP/Bluetooth connection, while hiding the heterogeneity of the used devices from the upper layers.

The *vi* also holds an array of possible outgoing interfaces in a neighboring database (NDB), similar to the Linux bridge's forwarding database. An entry contains a timestamp and is created upon receiving the first packet (i.e. a routing broadcast message or a route reply) of the associated neighbor/interface pair. Every consecutive incoming packet refreshes the timestamp. With this information the *vi* has a view of all neighbor nodes and the interfaces that are available to be used.

The vi collects information from the priority table to understand select a set of interfaces for communication, according to each device's needs. It is responsible for deciding on handovers and to perform them, switching the traffic from one interface to another using a simple timer.

3.1.2 Virtual Bandwidth Aggregation (VBA) / Decider

The VBA/decider, presented in Figure 3.2, has information concerning the interfaces that can be used from the *vi*, and according to that information, it chooses how the data to be sent, is divided between those interfaces. This is the mechanism that will increase the total throughput of the device, in comparison with a standard solution, since we are dynamically allocating the data we want to send between the existing interfaces.

Based in the number of physical interfaces present in a mobile device, their priority and RTT, the VBA/decider, decides what to do, in this case which interfaces should be used. Also the VBA is also able to monitor the mobile devices power levels and if necessary active a power saving mode that will use the interfaces with the lowest energy consumption to send the data.

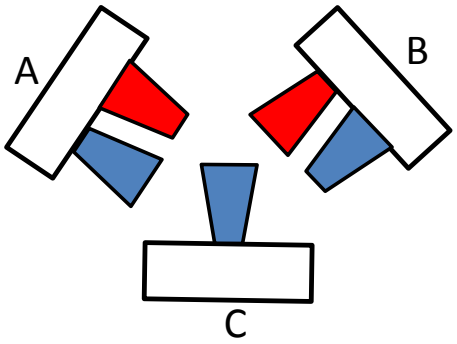
The implementation details, and how the decision is taken by the VBA, are explained afterwards, in section 3.2.

3.1.3 Priority Table

The priority table specifies a ranking among the interfaces, meaning that whenever a given neighbor can be reached through several interfaces, the interface with the highest priority is taken, being 0 de highest.

The default priority is also 0, which means if the user wants a specific interface to be used, he has to manually define the priority of each interface. If there are interfaces with the same highest priority, all of those can be used simultaneously, since no limitations were set by the user (e.g. no preference between Wi-Fi over Bluetooth).

The information gathered by this table is used by the *vi* to choose which interfaces should be used, as shown in Figure 3.3. In this example there are 3 nodes (A, B and C), node A has 2 interfaces (Bluetooth and Wi-Fi), node B has also 2 interfaces (Bluetooth and Wi-Fi) and node C has only a Bluetooth interface. Each node has a defined priority to each interface and an entry in the *Neighboring Database* (NDB) for each existing neighboring node. In this example, the node A, has the same priority for each interface, which means, if both are available, they can be used simultaneously.



	Priorities	ndb
A	Bnep0 – 1 Eth1 – 1	B – eth1, bnep0 C – bnep0
B	Bnep0 – 1 Eth1 – 2	A – eth1, bnep0 C – bnep0
C	Bnep0 – 1	A – bnep0 B – bnep0

Figure 3.3: The neighboring database (NDB), and the priorities corresponding to each node.

3.1.4 RTT Estimator

This block is responsible for estimating the average RTT for each physical network interface, which will then relay it to the VBA so it can decide which interfaces to use. We use the information present in the ACK packages reaching the device to estimate the RTT. This information is also used by TCP to calculate the *Retransmission Timeout* (RTO), this means, the amount of time the sender will wait for a given packet to be acknowledged. The estimate is based on the traffic leaving the device, and it is done so we can have a clear image of the neighboring nodes, and if a certain path used by a physical network interface is congested or not. This estimate is made periodically and the values stored in a hash table, so that the VBA can easily access this information.

3.1.5 Data Flow - Main Blocks

In this section, before describing the implementation details, we explain the connections, inputs and outputs within the main blocks composing the *vi* module.

As seen in Figure 3.4 the Virtual Interface receives information from the application layer and also from the VBA, which is responsible for deciding which network interfaces should be used. The VBA gathers information from three blocks, the priority table, the neighboring database (NBD) and the RTT Estimator. With this information it will decide which interfaces are to be used.

The interactions within the blocks are described next, according with the numeration present in Figure 3.4 and assuming packets being sent.

1. Data coming from the application layer to the physical network interfaces is intercepted and temporarily stored by the virtual interface.
2. The VBA receives a request from the virtual interface to choose the physical network interfaces to be used in distributing the data intercepted by the virtual interface.
3. VBA receives the list of available interfaces and associated neighbors from the neighboring database (NDB). These neighbors entries are refreshed via the routing algorithm control messages which are periodically broadcasted by neighborhood nodes.
4. VBA receives the priority associated to each physical network interface from the priority table.
5. VBA retrieves the RTT estimation from the RTT Estimator for each available interface with the highest priority.

6. VBA decides which physical network interfaces should be used and relays this information to the virtual interface.
7. The virtual interface divides the data stored in a temporary buffer and distributes it between the physical network interfaces defined by the VBA.

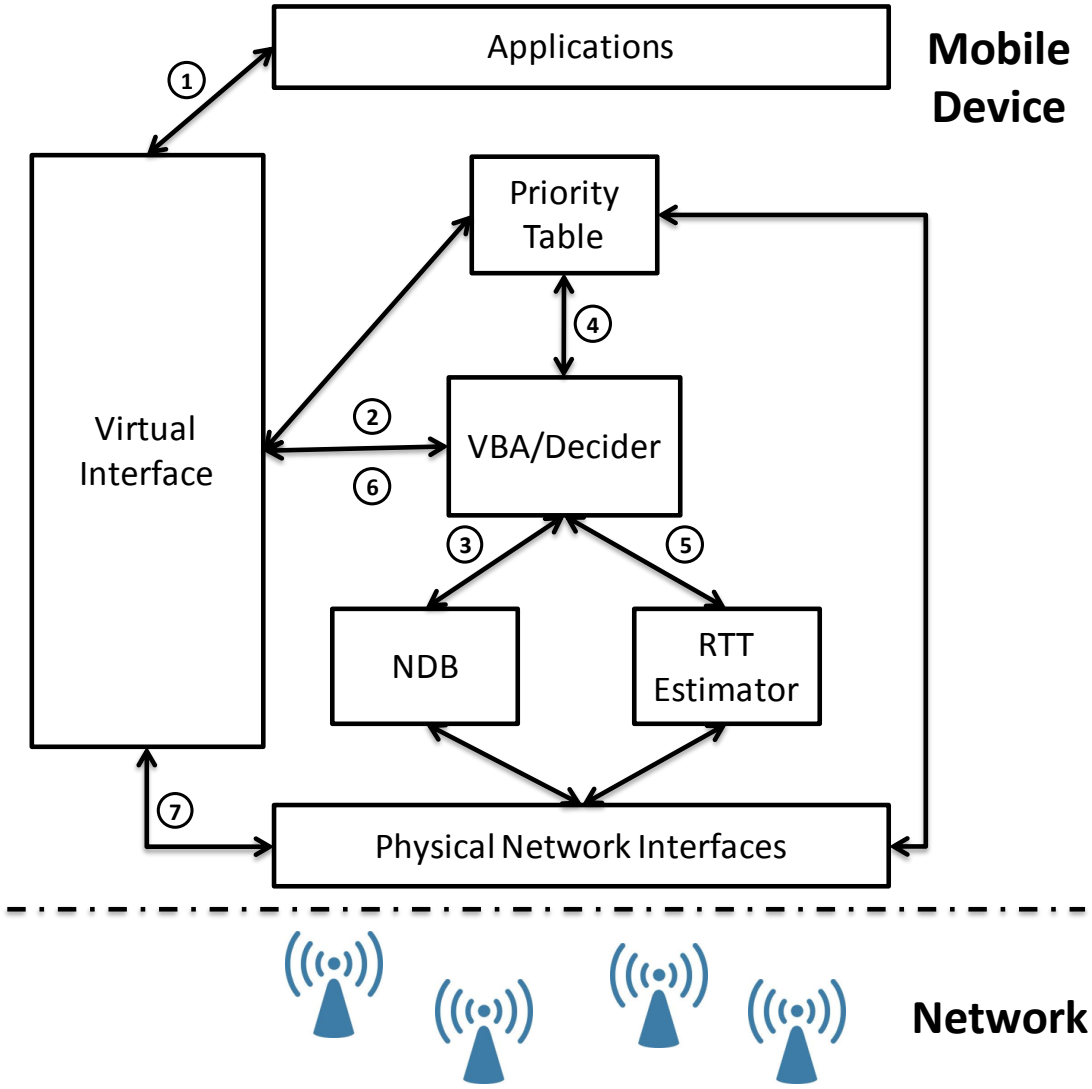


Figure 3.4: Data flow, explaining the interactions between the main blocks present in the *vi* module.

3.2 Implementation Aspects

The previous chapters described the thesis main building blocks, explaining the choices that were made in terms of tools to rely upon. This section is dedicated to the contributions of the thesis both in terms of concepts, implementation, and analysis. The section goes over specific

changes to parameter tuning, installation options and configurations, attempting to explain how each block introduced in section 3.1 was implemented.

The virtual network interface for transparent heterogeneous mobile ad-hoc networks in terms of implementation consists of three parts and can be seen in figure 3.5:

1. A kernel module providing the actual network interface;
2. A library providing programmatic access to the configurable options;
3. A user space utility to manage virtual interfaces.

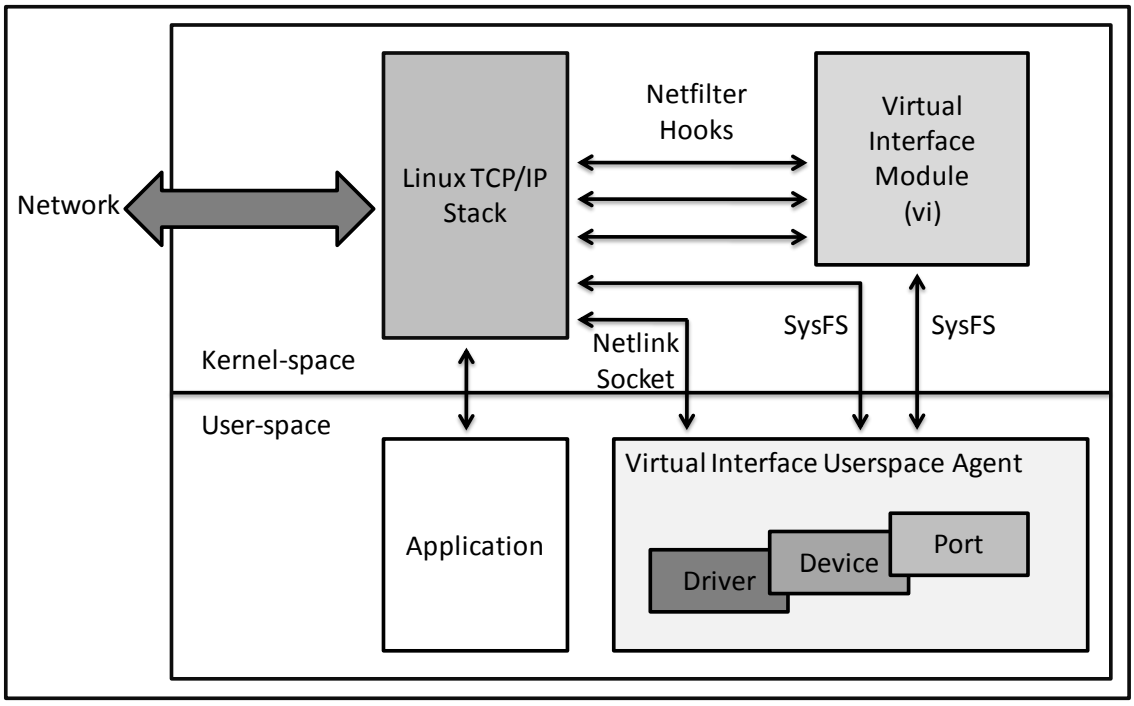


Figure 3.5: The Virtual Interface Architecture.

The kernel module contains all of the blocks introduced in the previous sections. The library and user space gives the user the possibility to configure each virtual interface parameter and properties according to their specific needs. This configuration, specific to the virtual interface is only available through *SysFS* (cf. Annex III). The data is intercepted and redirected via *Netfilter* hooks, which will be presented in the following sections.

3.2.1 The Kernel Module

The original work [28][32] followed an approach based on the bridge code [29], while this work, albeit taking a lot of inspiration out of it, started almost from scratch, since the previous version [32] had extensive limitations. The requirements for the module were as follows.

- Provide the functionality of the original virtual interface, namely:
  - Attach network devices.
  - Maintain a neighbor database.
  - Provide a mechanism capable of performing handover.
  - Provide a broadcast emulation.
  - Configuration via SysFS.
- No use of custom *ioctl*s.
- Possibility to use several physical interfaces simultaneously.
- Intercept the data using Netfilter hooks.
- Dynamically balance the data between the available physical priorities.
- Implementation should be as simple as possible.
- Minimize overhead and handover time.
- Ability to detect low battery levels and save energy by choosing which network devices to use.

In terms of implementation, we first needed to rewrite the previous implementation of the virtual interface code [28][32], since it was limited to a very specific version of the Linux kernel, and only worked with *sysfsutils v1.x* [34]. In version *2.x sysfsutils* suffered a number of changes to the way attributes were populated, another significant change was the removal of *struct sysfs_directory*, which rendered the previous module implementation non operational.

The second step was to improve the method used to intercept the data, since the previous one was too evasive [32]. The hook was placed in the general packet reception routine of a network device. Before passing the *sk_buff* to the upper layers it was checked if it has to be passed to a virtual interface. This previous solution added so much overhead to the *vi,* that the total throughput was significantly affected.

The solution we found was to insert *Netlfilter* hooks, introduced in the previous section (cf. section 2.4.1), removing the need to recompile the Kernel with the patch inserted into the *dev.c* file, substantially reducing the overhead added, as we will be seeing in Section 4.

The next step was to add a new block to the virtual interface, named Decider / Virtual Bandwidth Aggregation (VBA). This block is responsible for choosing which physical interfaces

to use from the ones behind the virtual interface. The VBA makes this decision based on three parameters, the priority and RTT of each interface and their availability according to the neighboring database, which contains the available neighbors and the path used to reach them.

Based on these three parameters, the VBA chooses how the data stored in the *dev_queue_xmit* buffer will be redirected to the available interfaces. For this purpose the physical interfaces are transparently aggregated under the virtual interface and a load balancing mechanism was implemented to distribute the data between the available interfaces. To do this we calculate the RTT of each interface, and use a simple function to calculate a value in the form of percentage, for each interface. This value defines the percentage of data intercepted by the *vi* a physical interface is responsible for. By doing this we are dynamically balancing the traffic between our physical interfaces, taking in consideration not only their RTT but also the paths actually being used.

The way we aggregate the interfaces under the virtual interface is the same used by the one implemented by the Linux bridge [29], where there is an aggregation of several interfaces, and the traffic is redirected between them. What was done was an adaptation of the mechanism used by Linux Bridge to our virtual interface (cf. section 3.2.1.4), so that it would also work in an ad-hoc network environment.

Additionally, the VBA is also able to monitor the device's power levels (the amount of battery left and if the device is plugged in to any power adapter), and if needed it will reduce the energy consumption by dynamically choosing the interfaces, based on their power consumption and throughput, making certain that the device uses the minimum amount of power to send the data. This extra function was also created from scratch, allowing the virtual interface to balance the data in a different way according to the power level, in order to save some energy.

The detailed implementation of each block will be presented in the next sections. A complete tutorial in how to install and use the virtual interface is also available in the annex section of this document (cf. Annex IV).

3.2.1.1 User interface

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

- Driver – /sys/bus/platform/driver/vi/
  Show version: version

Create a virtual interface: add
Remove a virtual interface: remove

- Device – /sys/class/net/vi<x>/vi/
  Attach physical network interface: add
  Detach physical network interface: remove
  Manipulate *maxdiff* value: Maxdiff
- Port – /sys/class/net/vi<x>/vi/ports/eth<y>/viport/ & sys/class/net/eth<y>/viport/
  Manipulate priority: priority

As we can, it is possible to create and remove any number of virtual interfaces, even though their names must start with “vi”, so it is possible to differentiate them from the remaining interfaces. We can add and remove the physical interfaces from a certain virtual interface and set their priorities, which will define the physical interfaces that are to be used.

In this section a new parameter that is associated with the *vi*, is also introduced, the so called *maxdiff* threshold. The *maxdiff* threshold unit is given by formula 3.1 and it decides how much two single entries within the neighborhood database may differ in terms of timestamps to keep the priority policy up. So a higher ranked interface entry is replaced by a lower priority interface if the timestamp differs for more than *maxdiff*. The 10ms default value has been considered due to the guidelines of the previous version [28], where the authors reached the conclusion that this was the value the *vi* would perform better with, in terms of throughput and handover time.

$$maxdiff = 10 \times \frac{Hz}{1000} \quad (3.1)$$

The other parameters were added to ensure the *maxdiff* value could vary according with each machine’s system timer frequency, which is represented in the equation by Hz, and by default is 1000 Hz [33].

3.2.1.2 Registering with the Device Model

The driver registers with the platform bus as there is no real bus it belongs to. The driver’s registration is necessary because there needs to be an interface to the driver in order to instantiate a virtual interface. This registration is made using the *SysFs*, described in Annex III; it contains device directories with links to the interface’s drivers.

The devices directory contains the global device hierarchy. This contains every physical device that has been discovered by the bus types registered with the kernel. It represents them in an ancestrally correct way, each device is shown as a subordinate device of the device that it is physically (electrically) subordinate to. Via this filesystem the user or system utilities can access and modify the parameters of devices and drivers, which is particularly useful in this situation, where we have a virtual interface.

The registration of a new interface is handled by a *sysfs* handler, using the function *register_netdevice(device)*, which receives as input the newly created virtual interface. To delete a virtual interface, we use the function *unregister_netdevice(device)* also from *sysfs*.

At initialization time, a device driver allocates a *net_device* structure and then initializes it with its necessary routines. One of these routines, called *dev->hard_start_xmit*, defines how the upper layer should enqueue an *sk_buff* for transmission. This routine takes an *sk_buff*. The operation of this function is dependent upon the underlying hardware, but commonly the packet described by the *sk_buff* is moved to a hardware ring or queue.

3.2.1.3 Data Interception

In order for the virtual interface to be able to receive the data coming to/from the application, it is necessary to intercept the data, which means we need to place a hook somewhere, to redirect the data to the virtual interface, and store it in a buffer.

As presented in [32], a hook into *dev.c* has shown to be very invasive and not at all flexible, causing a significant increase in the overhead added by the virtual interface. Another drawback of this solution is that the hook is still active even when the virtual interface is not being used, since the system is always checking if it should redirect the data to an existing virtual interface.

The most promising method we found to intercept the data, was using a custom *Netfilter* target. Such a target can be loaded and unloaded from kernel at any time. A well-understood architecture in the kernel and a userspace utility makes *Netfilter* a powerful tool. The *Netfilter* target for the virtual interface and other known *Netfilter* targets can also be combined in any favored way.

This hook performs exactly how the hook in *dev.c* does, but does not requires any modification to the Linux base files, and therefore there is no need to recompile the Kernel, with the modified version of the *dev.c* file, and it is only active when we are using the virtual interface module.

Packets will pass through hooking points sequentially. On each hooking point, it is possible to configure some filtering rules via the *iptables* command. After packets pass through NF_IP_PRE_ROUTING, the Linux kernel makes the routing decision to decide whether packets should enter the local processes or be routed to the next hop through the virtual interface and then redirected to a certain physical interface (this is decided by the decider, which will be described in section 3.2.1.7). In order to implement the *Netfilter* hooks, some major modifications were needed.

The *Netfilter* hook is created by the *net_hook* function, and registered using *nf_register_hook*. By adding the hook, we are intercepting and storing each data flow in a temporary buffer (*dev_queue_xmit*), while the virtual interface decides to which interface(s) it should be redirected to.

3.2.1.4 The Neighbor Database

The neighbor database is a hash table with the hash function calculated on the MAC address. A linked list for each hash value contains the entries corresponding to neighbors (cf. Figure 3.6 for a simplified representation of the neighbor database). The structure of a neighbor entry can be seen in listing 3.1.

```
01     struct net_vi_ndb_entry
02     {
03       struct hlist_node          hlist;
04       atomic_t                   use_count;
05       struct mac_addr            addr;
06       struct net_vi_port         *dst;
07       struct rcu_head            rcu;
08       unsigned long              ts;
09       unsigned                   is_local:1;
10     };
```

Listing 3.1: Structure net_vi_ndb_entry.

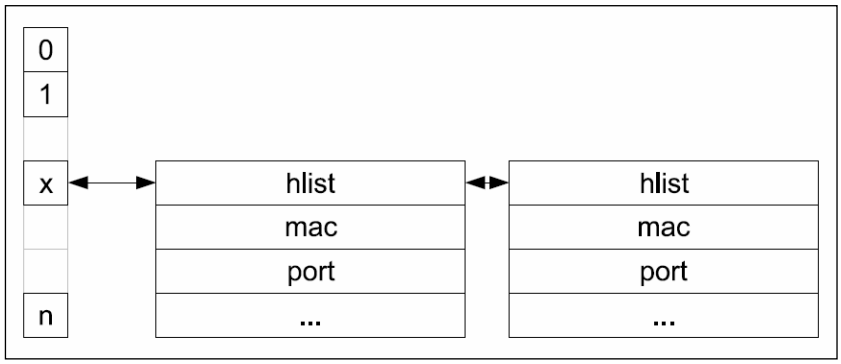


Figure 3.6: The neighbor database (simplified).

Let us now take a closer look at how routes get established in a heterogeneous mobile ad-hoc network when using virtual interfaces. In the case of a reactive routing protocol, it is the application that triggers a path setup. Since there is no route available yet, the routing protocol typically first broadcasts a route request. At the very beginning the neighboring database contains no entries but the transmission of a broadcast packet does not need any neighborhood information anyway. After the route request has passed several hops, a route reply eventually returns back to the origin. The route reply not only establishes the route but also creates an entry within the neighboring database, providing the *vi* with information on the interface to which the packets to the given neighbor have to be transmitted.

In the case of a pro-active routing protocol, things are slightly different. Here nodes periodically broadcast their neighboring information and therefore are also creating entries within neighborhood database. In both cases (proactive and reactive) the NDB entry is established in combination with the new route, regardless of whether the MAC technology changes or not.

The fields of this structure are used as follows:

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

Insertion

The function to insert and update entries into the neighbor database is the same. First, the hash table is searched for a matching entry. If one is found, it is updated; otherwise a new entry is created. The update sets the timestamp to the kernel time *jiffies*[1].

[1] A jiffy is the duration of one tick of the system timer interrupt. It is not an absolute time interval unit, since its duration depends on the clock interrupt frequency of the particular hardware platform.

Outgoing link selection

Outgoing links are selected according to the available neighboring nodes, present in the neighboring database. First we check if there is any available neighbor, if not, then the network interface cannot be used. After knowing which network interfaces can be used, the VBA decides which ones to use, based on their priorities and RTT estimation.

This structure is used to store the information of the available neighbors in an ad-hoc network, particularly the available nodes and which interface should be used to establish a connection with a certain node. We also added the possibility to use the virtual interface in a non ad-hoc scenario, which widens the possible scenarios the *vi* can be used in. This was done by adding the possibility to dynamically change the MAC of the virtual interface, so the packets coming from/to the *vi* would not be discarded, removing the necessity to have the Wi-Fi interfaces in promiscuous mode, which was a major drawback in the previous versions, since most of the Wi-Fi drivers does not support it.

3.2.1.5 Processing Incoming Packets

Incoming packets reach the virtual interface through the Netfilter hook. First, the sender's entry in the neighbor database is updated or created, and then the packet is passed up if:

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

Listing 3.2 shows the several checking conditions in a simplified manner, used to verify if the packets are to be passed up. If all of them fail, then the packets are dropped.

Receiving a packet is performed with *netif_rx*. When a lower-level device driver receives a packet (contained within an allocated *sk_buff*), the *sk_buff* is passed up to the network layer through a call to *netif_rx*. This function then queues the *sk_buff* to an upper-layer protocol's queue for further processing through *netif_rx_schedule*. Both dev_queue_xmit and *netif_rx* functions can be found in *linux/net/core/dev.c*.

```
01     /*
02       *  pass up all frames if virtual interface in promiscuous mode
03       */
04     if(vi->dev->flags & IFF_PROMISC) {…
05           vi_pass_frame_up(vi, skb)
06         …}
07
08
```

```
09      /*
10       *  pass up broadcast frame
11       */
12      if(dest[0] & 1){…
13            vi_pass_frame_up(vi, skb)
14          …}
15
16
17      /*
18       *  pass up local frame
19       */
20      if (mac_match(dest, vi->dev->dev_addr)) {…
21            vi_pass_frame_up(vi, skb)
22          …}
23
24
25      /*
26       *  pass up frames sent to one of the added interfaces (ports)
27       */
28      dst = __vi_ndb_get(vi, dest);
29      if (dst != NULL && dst->is_local) {…
30            vi_pass_frame_up(vi, skb)
31          …}
```

Listing 3.2: Handling incoming frames (simplified).

3.2.1.6 Processing Outgoing Packets

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

- Broadcast packet
  - Transmit over all attached interfaces;
- Normal packet;
  - Neighbor known: transmit over the corresponding outgoing link;
  - Neighbor unknown: transmit over all attached interfaces;

As we can see in Listing 3.3, there are two types of packets, broadcast and normal. Broadcasting is more frequent in ad-hoc networks than in wired networks, especially as the basic vehicle for on-demand route discovery. So, if the type is broadcast or we cannot find a certain neighbor in the neighboring database, the packet is transmitted over all attached interfaces. If not, then it is transmitted to a known neighbor over a certain outgoing link, present in the neighboring database, according with the policies set by the decider, which is described in the next section (cf. section 3.2.1.7).

To send a *sk_buff* from the protocol layer to a device, the *dev_queue_xmit* function is used. This function enqueues a *sk_buff* for eventual transmission by the underlying device driver (with the network device being defined by the *net_device* or *sk_buff->dev* reference in the *sk_buff*).

The dev structure contains a method, called *hard_start_xmit*, that holds the driver function for initiating transmission of a *sk_buff*.

This is a large structure containing all the control information required for the packet. Struct sk_buff has fields to point to the specific network layer headers:

- transport_header (previously called h) – for layer 4, the transport layer (can include TCP, UDP or ICMP header, and more);
- network_header – (previously called nh) for layer 3, the network layer (can include IP, IPv6 or ARP header);
- mac_header – (previously called mac) for layer 2, the link layer.
- skb_network_header(skb), skb_transport_header(skb) and skb_mac_header(skb) return pointer to the header.

```
01      /*
02              broadcast packet
03      */
04      static void vi_flood_deliver(struct net_vi *vi, struct sk_buff *skb)
05      {...}
06
07
08      /*
09              transmit packet. Net_device default interface
10      */
11      int vi_dev_xmit(struct sk_buff *skb, struct net_device *dev)
12      {...}
```

Listing 3.3: Handling outgoing frames (simplified).

3.2.1.7 Decider / Virtual Bandwidth Aggregation (VBA)

The VBA is responsible for choosing how the data, we want to send, is divided between the available interfaces. This is the mechanism, within our solution, that was created from scratch and shall increase the total throughput, in comparison with a basic setup, without the virtual interface, since we are transparently aggregating the available interfaces under the *vi* and dynamically allocating the data we want to send between the existing interfaces.

As mentioned before, there are several steps the *vi* must complete before choosing how to divide the data between the physical interfaces. First it is necessary to check three parameters:

- Priority;
- Availability;
- Round Trip Time (RTT).

Priority

To store the priority of each physical interface, we created a simple hash table that stores the names of each physical interface within a certain virtual interface, and their corresponding priorities. The access to the priority table is done in order do find all the physical interfaces with the highest priority, being 0 the highest (0 is also the default value for any physical interface that is added to a virtual interface). This is done using a simple function that accesses the priority table, and returns a list with all the highest "rated" interfaces. With this information we know which interfaces the user wants to prioritize.

Availability

The availability of each interface can be verified by using the information stored in the neighboring database, introduced in the previous section. In this hash table we have all the neighbor nodes, and which interfaces can be used to reach them. Since this table is constantly being updated by the routing protocol, we can use this information to check if the interfaces do in fact have any available path to the destination address.

This metric is verified so that one interface is not used when there are no available neighbors for it to transfer the data. In listing 3.4, we present a simplified version of the function used to check the neighboring database for a positive match. If a valid path is found for a specific physical interface it returns "0", if there is not any, the function returns "-1".

```
01  int vi_ndb_find(const struct net_vi *vi,const struct net_vi_port * port)
02      {
03              int I;
04              struct net_vi_ndb_entry *ndb;
05              struct hlist_node *h;
06              //rcu_read_lock();
07              for(I = 0; I < VI_HASH_SIZE; i++)
08              {
09                      hlist_for_each_entry_rcu(ndb, h, &vi->hash[i], hlist)
10                      {
11
12                              if (ndb->dst == port)
13                                                      {
14                                                       //rcu_read_unlock();
15                               return 0;
16                                                      }
17
18                      }
19              }
20                      //rcu_read_unlock();
21              return -1;
22
23
24      }
```

Listing 3.4: Availability of an interface.

TCP Round-Trip Time (RTT) estimation

After knowing which interfaces to use and their availability, it is necessary to calculate the *Round-Trip Time* (RTT) of each physical interface.

For this purpose and since TCP continuously estimates the current RTT of every active connection in order to find a suitable value for the retransmission time-out, we implemented a mechanism capable of calculating the RTT using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network.

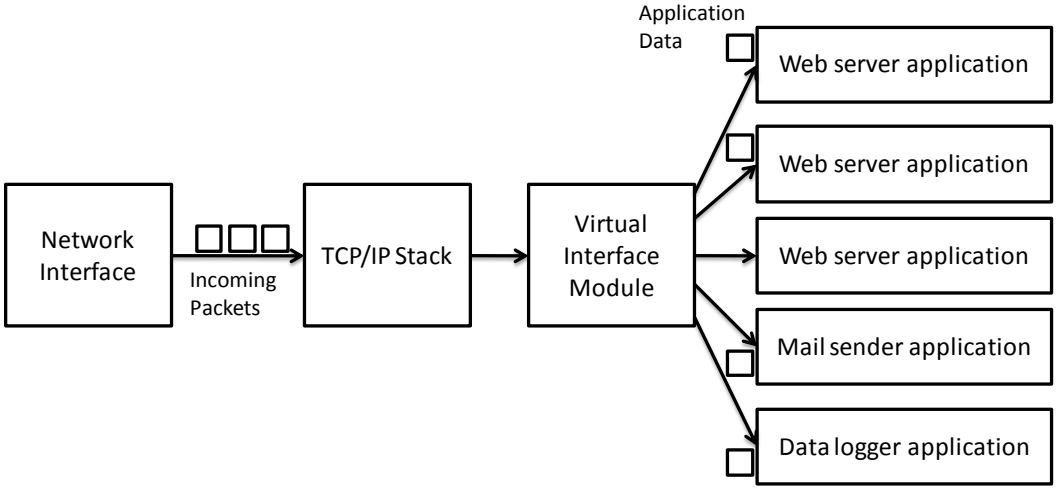


Figure 3.7: TCP/IP input processing.

For every data stream sent using TCP there is an acknowledge response that reaches the mobile device, these packets are intercepted by the Virtual Interface, which will then extract the RTT estimation. Figure 3.7 shows how packets enter the network device, pass through the TCP/IP stack, are intercepted by the Virtual Interface Module and then are delivered to the actual applications. In this example there are five active connections, three that are handled by a web server application, one that is handled by the e-mail sender application and one that is handled by a data logger application.

TCP implementations attempt to predict future round-trip times by sampling the behavior of packets sent over a connection and averaging those samples into a *Smooth Round-Trip Time* estimate (SRTT). When a packet is sent over a TCP connection, the sender times how long it takes for it to be acknowledged, producing a sequence of round-trip samples: S(1), S(2), S(3), …

With each new sample Si, the new SRTT is computed as [36][37]:

$$SRTT(i+1) = \alpha \times SRTT(i) + (1-\alpha) \times S(i) \quad (3.2)$$

Where *SRTT(i)* is the current estimate of the round-trip time, *SRTT(i+1)* is the new computed value, and **α** is a constant between 0 and 1 that control how rapidly the SRTT adapts to changes (usually **α**=1/8).

By applying formula 3.2 with the information extracted from the ACK packets constantly arriving, we are capable of estimating the average RTT values for each physical interface, without producing additional data [33]. This estimation is solely based on packets being transmitted by the applications located behind the virtual interface.

Additionally, the information regarding each interface's RTT is stored persistently in the format of a hash table, so that the VBA can easily access this information, and use it in function 3.3 to calculate the percentage assigned for each interface, which indicates how much data, within each traffic flow, the interface is responsible for.

Data division

The RTT estimation is only made from time to time (approximately each 5 seconds), but the function 3.3 which was created to balance the data in a proportionate way through the several physical interfaces, is executed for each data flow that is intercepted by the virtual interface.

$$P_{I_a} = \frac{1}{RTT_a \times \sum_{j=1}^{n}\left(\frac{1}{RTT_j}\right)} \quad (3.3)$$

The PI_a is the percentage a certain interface should be used to transfer the data intercepted by the *vi* and its value is between [0, 1]. The sum of all PI's must be one and it is calculated for every single available interface with the highest priority. The RTT is the Round-Trip Time of a certain interface and the summation interval is between 1 and n, being n the total number of available physical interfaces with the highest priority.

When a virtual interface is first created, and several interfaces are added, the table containing the results from formula 3.3 is empty. For this matter we use function 3.4, which uses the bandwidth from each interface, as a metric, to calculate the necessary proportions that will be used to calculate the amount of data each physical interface is responsible for, within a certain data flow.

Note that this is only temporarily; function 3.4 is only applied if the RTT metric fails. Meaning there is not enough available information regarding the RTT estimation of every physical interface.

$$P_{I_a} = \frac{Bandwidht_a}{\sum_{j=1}^{n}(Bandwidht_j)} \quad (3.4)$$

Again, the PI_a is the percentage a certain interface should be used to transfer the data intercepted by the *vi* and its value is between [0, 1]. The sum of all PI's must be one and it is calculated for every single available interface. The Bandwidth values are acquired via *SysFS* and the summation interval is between 1 and n, being n the total number of available physical interfaces. The values acquired by this function, are only used if there is at least one or more interfaces whose RTT values were not calculated yet, since the bandwidth used for this calculation is not the actual throughput (does not take into consideration, the path being used by the physical interfaces).

Load Balancing

After calculating all the PI's, the VBA will now redirect the data to the physical interfaces, taking into consideration the obtained values.

Note that each interface has an assigned percentage, and the sum of the percentages of all available interfaces is 100%. So if for example we have 10Mb to transfer and 2 available interfaces (eth0 and eth01), eth0 has a percentage of 20% (PI = 0,2) and eth1 has 80% (PI = 0,8), this means that eth0 will transfer approximately 2Mb while eth1 transfers 8Mb.

In comparison with the previous versions [28][32], where the authors only used priorities to divide the data between the physical interfaces, we are now using a dynamic load balancing mechanism since we are dynamically allocating the data through the existing interfaces. By implementing such mechanism we are also making sure that no bottlenecks are being created, since by taking into consideration the actual RTT value for each interface, we are not only analyzing each interface's throughput but also the path actually being used by all available interfaces with the highest priority.

To better explain all the methods VBA has to offer and how it distributes the data between the available physical interfaces, simplifying the whole implementation description, we can divide the available options into two modes of operation:

- Mode I – 1 Physical Interface

  There is only one interface with the highest priority, being 0 the highest. Only one interface is used at a certain time to send the data. If that interface goes down, then the VBA uses the interface with the second highest priority. If by any case the VBA cannot find any available path, in the neighboring database, then the virtual interface will transmit the data coming from the application via broadcast mode, through all the interfaces present in the device, to assure that it will reach its final destination;

- Mode II – 2 or more Physical Interfaces

  If there are several interfaces with the highest priority, all those interfaces are used, and a load balancing mechanism is applied to distribute the data trough all the available interfaces. If all interfaces go down, except one, then the VBA goes into Mode I. The load balancing mechanism, divides the intercepted data, stored in the buffer, between all the available interfaces. This mechanism is only applied to interfaces with the same priority, and the amount of data an interface is responsible for sending is calculated using either formula 3.3 or formula 3.4, depending if the table containing the RTT values is empty or not.

Energy Consumption

The VBA is also responsible for monitoring the energy power levels, foreseeing the necessity of saving energy, by using the best suited interfaces. We do this by accessing the files present in the */proc/acpi/battery* directory, which stores information regarding the actual battery status of the device. We extract the power level and information that tell us if the device is connected or not to any power adapter. This information can also be accessed parsing the information returned by the command *acpi –a*.

If the battery level is below 10% and if the mobile device is not plugged in to any power adapter then the power saving mode is activated. This mechanism is described in the next section.

3.2.2 Power Saving Mode

This mode as mentioned before is only activated if the battery level is below 10% and if the mobile device is not plugged in to any power adapter. The VBA is responsible for monitoring both values, and will activate this mode to insure, that the data will be transferred while

consuming the minimum amount of power possible. This verification is made by de VBA every 120 seconds.

The function used to distribute the data between the physical interfaces is similar to the one presented in the previous section (formula 3.3). It takes into consideration the RTT values, in order to extrapolate the throughput and the energy consumption of each interface. Based on these two parameters we find the solution that consumes the least amount of energy to send the data.

For example, if we have one interface with a lower energy consumption than the remaining, that does not mean it will consume less energy to send a certain data flow. We have to take in consideration its throughput, verify how long it will take to transfer the data and during that time, how much energy those interfaces will consume.

Throughput

The Throughput is measured in bits per second, it is estimated based on the RTT measurements and it is calculated using formula 3.5. Note that by default the TCP Buffer size >= TCP Window size. Typical TCP window size is equal to 64 Kbyte, and the RTT is measured in seconds [38].

The value we obtain in formula 3.5 is a theoretical value of the throughput. It is calculated in order to estimate the energy consumption of a certain interface and is used in formula 3.6. To simplify the calculation we are assuming a packet loss of 0%, since the obtained values are merely for comparison reasons, so we can understand which interfaces use the most amount of energy to send a certain data flow.

$$Throughput = \frac{TCP\ buffer\ size}{RTT} \quad (3.5)$$

Energy Consumption per packet

When a node sends or receives a packet, the associated network interface, decrements the available energy according to the following parameters: (a) the specific network interface controller (NIC) characteristics, (b) the size of the packets and (c) the bandwidth used. The following formula represents the energy used (in Joules) when a packet is transmitted or received (Formula 3.6) and the packet size is represented in bits [38]:

$$Energy_{tx,rx} = \left(\frac{Energy\ Consumption * Energy\ Supply * PacketSize}{Throughput}\right) \quad (3.6)$$

The energy consumption is measured in *miliamperes* (mA), varies with the interface being used and if a packet is being transmitted or received. The energy supply also varies with the device being used and is measured in Volts (V).

Although the equipment consumes energy, not only when sending and receiving but also when listening, we have assumed in our model that the listen operation is energy free, since all the evaluated ad-hoc routing protocols will have similar energy consumption due to the node idle time.

Energy Consumption per data flow

After knowing how much energy a network interface requires for sending a packet, we can now calculate if the current set up, defined by the VBA is consuming the least amount of energy to send a certain data flow. For that we use formula 3.7, representing the energy consumed during the transmission of the data present in the output buffer (in Joules). The *BufferSize* and *PacketSize* are both represented in bits and the summation interval is between 1 and n, being n the total number of available physical interfaces with the highest priority. The PI represents the value calculated in either formula 3.3 or formula 3.4 and *Energy* represents the energy used (in Joules) when a packet is transmitted, and it is calculated in formula 3.7 [38].

$$Energy\ Consumed_{VBA} = \frac{BufferSize}{PacketSize} \times \sum_{j=1}^{n}\left(Energy_j \times PI_j\right) \quad (3.7)$$

Now we need to compare the acquired energy consumed value with the energy the interface with the lowest energy consumption would require for sending the same amount of data. For that we use formula 3.8. The parameters are the same as the ones in formula 3.7, but now we are only taking in consideration one interface, not all the interfaces present in the mobile device.

$$Energy\ Consumed_I = \frac{BufferSize}{PacketSize} \times Energy_I \quad (3.8)$$

After acquiring this second value we compare both energy results, and verify if $EnergyConsumed_{VBA} \geq EnergyConsumed_I$. If this is the case, then the VBA will only use the interface with the lowest energy consumption to transmit the data flow, since it will consume less energy.

3.2.3 The libvi library

This library provides the interface given in listing 3.5. The library uses *libsysfs* [42] to access the virtual files in the SysFS to configure the virtual interface. Before actually using the functions provided by *libvi* it has to be initialized by calling *vi_init*. This is one of the two possible ways to configure the virtual interface; the second one is presented in the next section.

```
01      int vi_init(void)
02      {
03              vi_class_net = sysfs_open_class("net");
04              return 0;
05      }
06
07      int vi_addvi(const char *name);
08
09      int vi_delvi(const char *name);
10
11      int vi_addif(const char *vi, const char *ifname);
12
13      int vi_delif(const char *vi, const char *ifname);
14
15      int vi_set_portpriority(const char *ifname, unsigned long prio);
16
17      int vi_set_maxdiff(const char *vi, unsigned long maxdiff);
18
19      int vi_get_portpriority(const char *ifname, unsigned long *prio);
20
21      int vi_get_maxdiff(const char *vi, unsigned long *maxdiff);
```

Listing 3.5: Virtual interface configuration library.

3.2.4 The victl command

The *victl* command is a command-line utility which uses *libvi* to manage virtual interfaces. If *victl* is run without parameters it displays a help message which explains how to use it. Listing 3.6 shows the available commands showed upon running the command *victl* without parameters:

- addvi - Create a virtual interface;
- delvi - Remove a virtual interface;
- addif - Attach physical network interface to an existing virtual interface;
- delif - Detach physical network interface from an existing virtual interface;
- setmaxdiff - Manipulate *maxdiff* value;
- setportprio - Manipulate priority from a certain physical interface.

```
[root@mota]# victl
commands:
help command list
addvi <vi> add vi
delvi <vi> delete vi
addif <vi> <device> add interface to the vi
delif <vi> <device> del interface from the vi
setmaxdiff <vi> <maxdiff> set maxdiff
setportprio <vi> <port> <prio> set port priority

[root@mota]# victl add vi0
[root@mota]# victl addif vi0 wlan0
[root@mota]# victl setprio vi0 wlan0 1
[root@mota]# victl addif wlan1
[root@mota]# victl setprio vi0 wlan1 1
[root@mota]# ifconfig wlan0 0.0.0.0
[root@mota]# ifconfig wlan1 192.168.2.1
[root@mota]# victl setdiff 200
[root@mota]# ifconfig

wlan0 Link encap:Ethernet HWaddr 00:02:72:B2:78:D2
UP BROADCAST RUNNING MULTICAST MTU:1500
RX packets:0 errors:0 dropped:0 overruns:0
TX packets:4 errors:0 dropped:0 overruns:0
collision:0 txqueuelen:100
RX bytes:104 (104.0 b) TX bytes:88 (88.0 b)

wlan1 Link encap:Ethernet HWaddr 00:02:2D:7B:88:D1
UP BROADCAST RUNNING MULTICAST MTU:1500
RX packets:1093 errors:277 dropped:0 overruns:0
TX packets:51 errors:0 dropped:0 overruns:0
collision:0 txqueuelen:100
RX bytes:65778 (64.2 Kb) TX bytes:11386
Interrupt:11 Base address:0x100

vi0 Link encap:Ethernet HWaddr 00:02:72:B2:78:DC
inet addr:192.168.2.1 Bcast:192.168.2.255
UP BROADCAST RUNNING MULTICAST MTU:1500
RX packets:0 errors:0 dropped:0 overruns:0
TX packets:0 errors:0 dropped:0 overruns:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

Listing 3.6: The victl command.

By running the command *dmesg* it is possible to debug the module, and see step by step what actions is the virtual interface executing. In listing 3.7 we have an example, were two interfaces, *wlan0* and *wlan1* were added to the virtual interface *vi0*. In this example we can clearly see the *vi* finding a match in the neighboring database and using both *wlan0* and *wlan1* to send and receive the data coming from and to the application layer.

```
01 [ 2406.936061] debug.vi: sending interface index=0
02 [ 2406.936066] debug.vi: deliver packet through wlan1
03 [ 2406.936560] debug.vi: handle_frame
04 [ 2406.936566] debug.vi: entering vi_ndb_insert
05 [ 2406.936570] debug.vi: entering ndb_insert
06 [ 2406.936574] debug.vi: ndb_insert, hash: 147
07 [ 2406.936578] debug.vi: ndb_insert looking for existing entries
08 [ 2406.936583] debug.vi: ndb_insert, found matching entry
09 [ 2406.936588] debug.vi: ndb_insert, update existing entry
10 [ 2406.936592] debug.vi: ndb_insert, done
```

```
11 [ 2406.936595] debug.vi: leaving vi_ndb_insert
12 [ 2406.936599] debug.vi: handle_frame_finish
13 [ 2406.936603] debug.vi: passing up local frame
14 [ 2406.936608] debug.vi: pass_frame_up from wlan1 to vi0
15 [ 2406.936612] debug.vi: frame passed up
16 [ 2406.936616] debug.vi: leaving handle_frame_finish
17 [ 2406.936636] debug.vi: vi_loading_blance_policy return sending interface index=0
18 [ 2406.936641] debug.vi: sending interface index=0
19 [ 2406.936645] debug.vi: deliver packet through wlan0
20 [ 2407.942201] debug.vi: vi_loading_blance_policy return sending interface index=0
21 [ 2407.942209] debug.vi: sending interface index=0
22 [ 2407.942214] debug.vi: deliver packet through wlan0
23 [ 2408.950183] debug.vi: vi_loading_blance_policy return sending interface index=1
24 [ 2408.950191] debug.vi: sending interface index=1
25 [ 2408.950196] debug.vi: deliver packet through wlan1
26 [ 2409.061191] debug.vi: vi_loading_blance_policy return sending interface index=1
27 [ 2409.061197] debug.vi: sending interface index=1
28 [ 2409.061202] debug.vi: deliver packet through wlan1
29 [ 2409.156279] debug.vi: vi_loading_blance_policy return sending interface index=1
30 [ 2409.156286] debug.vi: sending interface index=1
31 [ 2409.156291] debug.vi: deliver packet through wlan1
```

Listing 3.7: Debugging the virtual interface module.

3.3 Limitations

During the implementation and testing of the *vi* module, some limitations were found, unfortunately due to time constraints we could not solve them all, the ones that still apply are presented next.

The first identified problem was that the *vi* module will only work if the access points to which the network interfaces are connected to, are in the same network, since the virtual interface can only have a single IP address at a given time. So the usage of this module is limited to an ad-hoc scenario or to a situation where there are several APs, all under the same network (e.g. Campus University network).

Another limitation that was identified is that by adding the hooks to intercept the data, we are also adding a regular timer tick, The timer tick is a timer interrupt that is usually generated HZ (Hertz) times per second, with the value of HZ being set at compile time and varying between around 100 to 1500. Running without a timer tick means the kernel does less work when idle and can potentially save power because it does not have to wake up regularly just to service the timer, and since we are adding such interrupt, this means the kernel will not go into idle, and will not be able to save as much energy as it would. Figure 3.8 shows what processes/drivers are keeping the mobile device active. As we can see the *vi* module is one of the main causes for wakeups (16.2%), taking a toll in the mobile devices energy savings, when in idle.

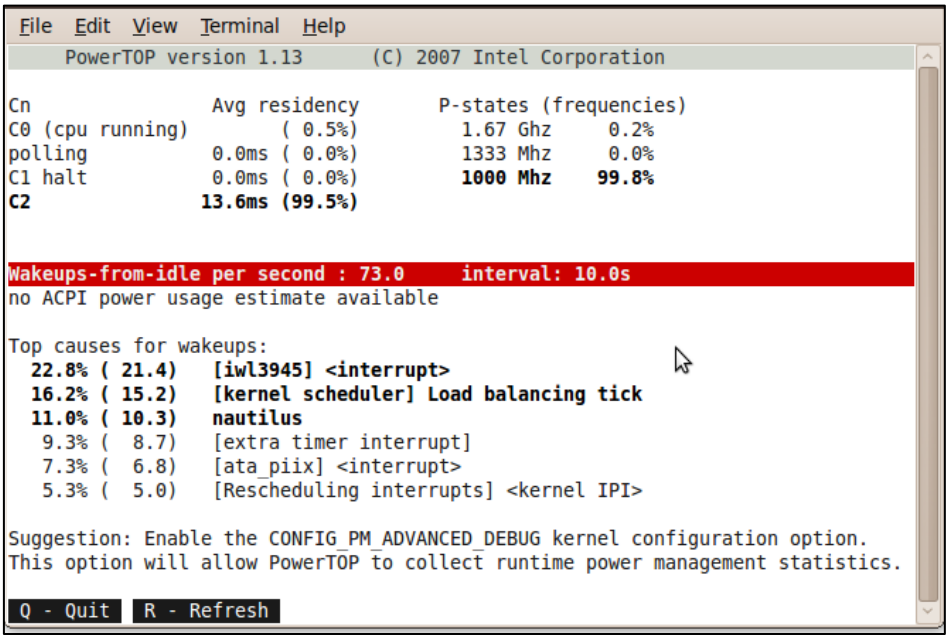


Figure 3.8: Kernel main causes for wakeups, measured with PowerTop[1].

We considered that the energy consumption of a certain interface when in idle time is zero, which is not actually true. For a more correct approach the energy a certain interface is using when in idle, should be taken in consideration in the used algorithm. It would be interesting to make such modification and compare the results, so that we could understand if there is actually any impact in the amount of energy saved by the power saving mechanism implemented within the *vi* module.

Finally, the *vi* module will only work with physical interfaces using the following network standards:

- IEEE 802.11/WLAN;
- IEEE 802.15/Bluetooth;
- IEEE 802.3/Ethernet.

[1] PowerTop, http://www.lesswatts.org/projects/powertop/

3.4 Security Concerns and Other Aspects

The broadcasting nature of transmission and the nodes self routing environment opens up the perception of security in ad-hoc networks. The security issue of ad-hoc is of large concern taking into account its various factors like its open network, mobility factor and other factors. In this section we address some of this security concerns and since the security behind the module is out of scope of this thesis, we merely identify possible issues and propose some solutions intended to solve them.

The first identified problem is related with the way we estimate RTT. The RTT estimation is made based on the data that is entering the device via the available network interfaces, if there is an attacker placed between the device and a certain AP, he could alter the values present in the ACK packets used by TCP to estimate the RTT, which will then affect the way we choose the physical interfaces. The RTT estimator will take the RTT information out of the packets entering the mobile device, and since these values were altered by the attacker, we are actually basing our decisions in values that are not reliable.

The second attack that was indentified may also affect the decision of the VBA. If an attacker is able to fake the periodically messages sent by the routing protocol, which we use to update the neighboring database, then the VBA will think that interface is active. This attack will cause the virtual interface to send data to physical network interfaces that have no available neighbors.

Both identified attacks can be performed by the method known as Man-In-The-Middle, where attackers intrude into an existing connection to intercept the exchanged data and inject false information. It involves eavesdropping on a connection, intruding into a connection, intercepting messages, and selectively modifying data.

In order to provide solutions to the security issues involved in ad-hoc networks, we must elaborate on the two of the most commonly used approaches in use today:

- Prevention
- Detection and Reaction

Prevention dictates solutions that are designed such that malicious nodes are thwarted from actively initiating attacks. Prevention mechanisms require encryption techniques to provide authentication, confidentiality, integrity and non-repudiation of routing information. Among the existing preventive approaches, some proposals use symmetric algorithms, some use asymmetric algorithms, while the others use one-way hashing, each having different trade-offs and goals.

Prevention mechanisms, by themselves cannot ensure complete cooperation among nodes in the network. Detection on the other hand specifics solutions that attempt to identify clues of any malicious activity in the network and take punitive actions against such nodes

The first attack can be mitigated by measuring the RTT using a different approach. Instead of basing our RTT estimation in the TCP traffic entering the device, we can create and send ICMP packets via all network interfaces. The messages being sent must be encrypted and signed, so that we can verify their integrity and authenticity. This method will give us a more correct reading in terms of RTT but we are also adding additional overhead. Other problem that might also occur with this solution is firewalls blocking ICMP packets, being this the main reason why we opted for the lesser secure but more efficient method of estimating the RTT.

One way of lessen the impact of the second attack, is to periodically send encrypted packets (for example crypto puzzles) to all available neighbors and wait for a response. A crypto puzzle is a quickly computable cryptographic problem formulated using the time, a server secret, and additional client request information. In order to have server resources allocated to it for a connection, the client must submit to the server a correct solution to the puzzle it has been given. If we get a response from a certain neighbor it means that the interface has in fact available neighbors to which it can send the data.

4 Performance Evaluation

This chapter is dedicated to the performance evaluation of the main building blocks of this thesis, attempting to answer the questions that lead to this work and that can be aggregated into three main aspects:

- Is the overhead added by the virtual interface excessive?
- Are the implemented mechanisms improving the total throughput? If yes, in which situations?
- Is the power consumption mechanism, present in the *vi* module, saving any energy?

Answers to these questions are provided by relying on several experimental sets, where the number of access points, interfaces, and also the type of data flows was varied.

The neighbor database look-up and the detour the packets have to take naturally impair throughput. So, the performance of the virtual interface was measured according to throughput, overhead time and also Handover-time in case of a vanishing link, since packets can get lost.

The chapter starts by detailing the goals for the experiments, and the followed methodology. A generic description of the evaluation parameters and scenarios is then provided, followed by a description of the topologies implemented, and of the traffic settings as applied to the experiments. Section 4.2 explains in a detailed manner the results obtained during the experiments and which refer to packet loss, energy consumption, as well as end-to-end delay and total throughput. Finally, section 4.3 summarizes the results obtained in the test experiments and answers the questions presented above.

4.1 Evaluation Objectives and Settings

The experiments presented in this section have as main goal to analyze the virtual interface vs. normal scenario (without virtual interface) in terms of end-to-end delay, packet jitter, as well as packet loss and total throughput. For a specific IP datagram x, the end-to-end delay $d(x)$ is defined as the time it takes for the packet to travel from source to destination, i.e. the interval between the time the packet was sent and the time it was received [4], as in equation 4.1.

$$d(x) = t_r(x) - t_t(x) \quad (4.1)$$

Where $t_r(x)$ and $t_t(x)$ represent the time at which packet x was received and transmitted, respectively.

The inter-packet delay is defined as the variation between the delays of two consecutive packets. For two consecutive packets, x and y, x being the first packet received, the inter-packet delay is defined in equation 4.2.

$$D(x) = d(y) - d(x) \quad (4.2)$$

Packet loss, P, is here defined as the ratio between the number of lost packets and the number of packets that were sent, not counting the packets received out-of-order. This computation is performed based on the sequence number of the packets received. For every flow, the expected packet sequence number is kept, and if the received sequence number is higher than the expected, then the total number of lost packets is incremented. Packet loss is expressed in percentage, according to equation 4.4.

$$P = \frac{L}{N} \times 100 \quad (4.4)$$

Where N is the number of total packets that were sent by the source node and L is the number of packets that did not reach the destination node.

Throughput, T, is here defined as the ratio between the RCV buffer size and the RTT (Round-trip time) [37]. Hence, total throughput is calculated as described in equation 4.5. Throughput is expressed in Mbps, the RTT in seconds and the buffer size in Megabits.

$$T = \frac{RCV\ buffer\ size}{RTT} \quad (4.5)$$

Energy consumption, given by W, is here defined as is the rate at which work is done when one ampere (A) of current flows through an electrical potential difference of one volt (V). Energy consumption is expressed in milliwatt hour (mWh), according to equation 4.6.

$$W = V \times A \quad (4.6)$$

A milliwatt-hour is the amount of energy equivalent to a steady power of 1 milliwatt running for 1 hour.

4.1.1 Traffic and Network Settings

The traffic used in the simulations was generated by relying on *iperf*[1], since it is supported by both Linux and Windows operating systems, via its graphical component *jperf*[2]. It is more focused on measuring the network available bandwidth, capable of measuring bandwidth and datagram loss, it also presents the results of jitter and RTT. Additionally it is also possible to specify a traffic type, TCP or UDP, although it is not possible to specify the traffic pattern. To evaluate the overhead and handover time, we used the command ping.

For each test, in terms of measuring the available bandwidth we also tested different packet sizes, which are important to determine the per-packet overhead. The values are averaged over 20 readings and each reading takes 100 seconds to acquire (except the handover time that requires only the number of packets lost while handover is occurring).

All readings were taken on a laptop Sony Vaio PCG-7N2M. A Tsunami desktop computer using Windows Vista operating system was also used as server. The access points used in the experiments were routers Fonera+[3], flashed with the Linux based firmware DD-WRT[4].

- Sony Vaio PCG-7N2M
  - Memory: 1GB
  - Intel® Centrino Core™ Duo T2300 Processor 1.66 GHz
  - Ubuntu, Linux 2.16.31.14
  - Virtual Interface Module v2.0
  - Wi-Fi interface 1: Integrated wireless 802.11a/b/g
  - Wi-Fi interface 2 (usb): D-Link DWL-G122 High Speed 2.4GHz (802.11a/b/g)
  - Wi-Fi interface 3 (usb): D-Link DWL-122 (802.11b)
  - Iperf client

- Tsunami Desktop
  - Memory: 3GB
  - Intel® Core™ i7 CPU 920 @ 2.67GHz
  - Windows Vista™ Home Premium 32bits
  - Onboard gigabit LAN interface
  - Iperf Server

---

[1] Iperf, http://sourceforge.net/projects/iperf/

[2] Jperf, http://sourceforge.net/projects/jperf/

[3] Fonera, http://www.fon.com/

[4] DD-WRT, http://www.dd-wrt.com

Our solution was tested with AODV-UU [43] routing protocol and different types of data. The obtained results were of course compared with a simple scenario with no virtual interface, no bandwidth aggregation and no load-balancing mechanisms, referred as RAW in the following section. The detailed testbed configuration used for the Fonera+ routers, can be found in the annex section of this thesis.

The tests were done in both an ideal environment, where there was no overload of the APs, and in a saturated environment, where we have several different users using the ad-hoc network simultaneously, lowering the throughput of each individual node. This is an interesting test scenario, that was considered in order to find out the different situations for the *vi* to perform better.

4.1.2 Main Topologies

The experiments run considered three different topologies as basis for the different developed scenarios. In each topology we test both the load-balancing mechanism as well as the power saving mode, in terms of throughput and delay, to understand the actual impact of the virtual interface. The first topology considered (Topology I) is illustrated in Figure 4.1. Topology I serves as a control test, as it only contemplates one access point and one physical network interface present in the mobile device A.

For the mentioned topology, node A is connected by means of an ad-hoc network. The purpose of this configuration is related to the need to get data that serves as control, relating to the situation where we are not using the virtual interface, which will then be compared with more complex scenarios where we use the *vi*. We also use this topology to estimate the delay added by the virtual interface, in order to understand the impact the *vi* is causing in terms of throughput in a situation where only one single node is available. The virtual interface will be situated behind the physical interface, as presented in Figure 4.1.

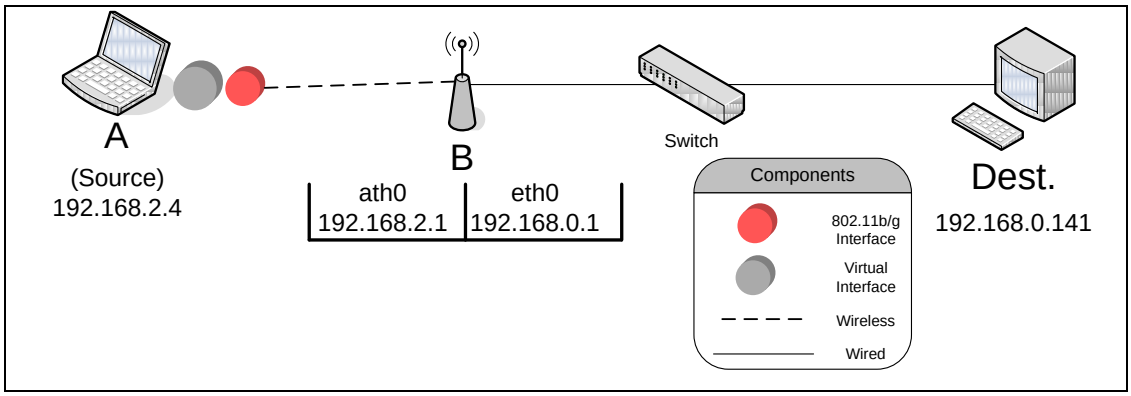


Figure 4.1: Topology I, one network interface and one AP.

The second considered topology (Topology II) is represented in Figure 4.2. Topology II is in fact similar to I, being the only difference the number of physical network interfaces present and the number of access points, which was now increased to 2. Such increment will assist in trying to understand the impact caused by the number of interfaces in the performance evaluation of the virtual interface.

What we pretend to test with this topology is how the extra access point will impact the usage of the *vi* in terms of total throughput and overhead, since in this case a new path will be available, which should cause a slight increment in terms of overhead. Two tests will be made using this topology, in the first one, both access points will only have user A connected and transmitting data, while in the second test, the access point B will be saturated with data from a third party computer. Both results will then be compared, in order to understand how the vi will react to this change in the topology.

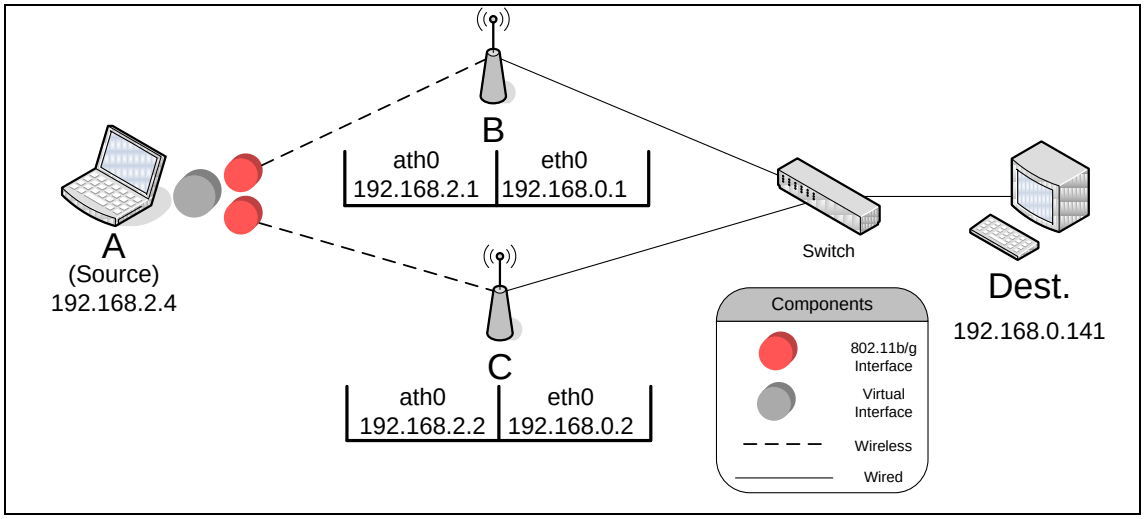


Figure 4.2: Topology II, two network interfaces and two APs.

Finally topology III is illustrated in Figure 4.3. It still consists of paths that are one hop long, but now there is just one AP and one more physical network interface in comparison with topology II. In this experiment the AP is saturated with data from a secondary source. Our expectations were to observe what would be the reaction of the *vi* when there is an increment in the number of interfaces, in a worst case scenario where there is just one AP and multiple network interfaces. For this scenario we also used different network interfaces, two IEEE 802.11g and one IEEE 802.11b.

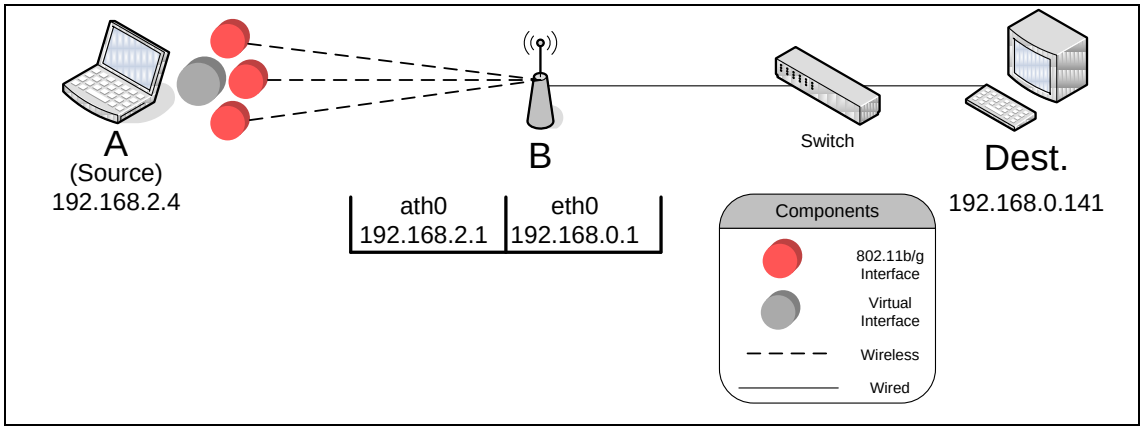


Figure 4.3: Topology III, three network interfaces and one AP.

4.2 Evaluation Results

In this section we present the experimental results. For each scenario run, results concerning packet loss, delay, and throughput as well as power usage are presented and explained.

Throughput was measured using *jperf* as mentioned in the previous section. This is a client-server based tool, which can measure both TCP and UDP traffic. *Raw* measurements were taken without the virtual interface. The varying packet sizes are relevant to determine the per-packet overhead. The values are taken during 100 seconds, averaged over 20 readings and show the throughput between two nodes in Mbps. To evaluate the delay and handover time added by the *vi* module, we used the command *ping*.

Finally, to measure the total energy consumption during a certain period of time, we created a bash script that extracts to a file the amount of energy in miliwatt hour (mWh) the mobile device has. The bash script can be found in Annex VI.

4.2.1 Experiment 1

The first experiment relies on Topology I (cf. section 4.1.2) which represents a simple topology, since we want to test the difference of performance between the *vi* and the raw measurements taken without it. In the topology, node A corresponds to a sender with just one physical interface and node Dest. corresponds to the destination. The sender generates traffic according to the settings described in section 4.1.1 and packet sizes vary as 1 Kbyte, 2 Kbytes,

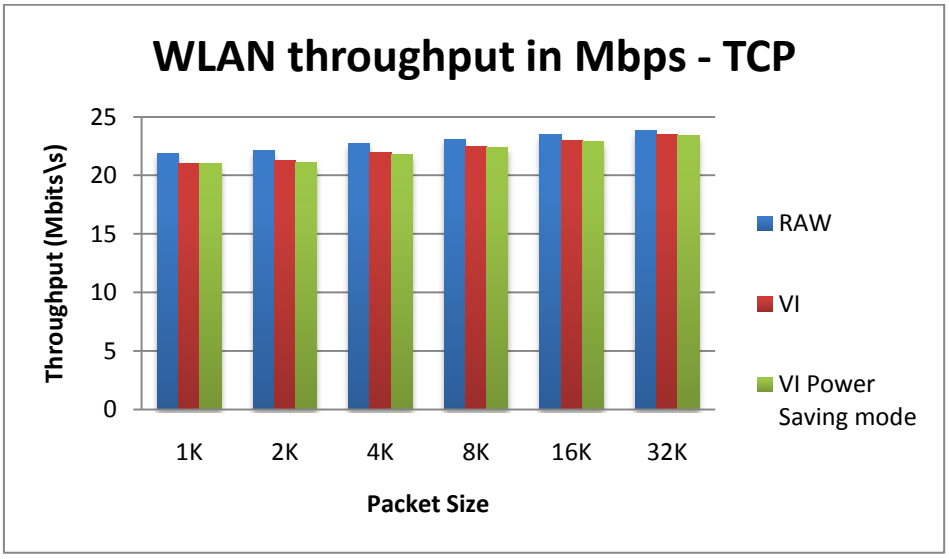
4 Kbytes, 8 Kbytes, 16 Kbytes and 32 Kbytes, in order to allow measuring the per-packet overhead.

Throughput

Starting by the analysis concerning the achieved throughput using TCP data, Table 4.1 shows the throughput and corresponding standard deviation (**σ**) when using the virtual interface, and the raw measurements taken without it. Graph 4.1 plots that information, so it is possible to make a better comparison between both situations.

	1KB	STD (σ)	2KB	STD (σ)	4KB	STD (σ)	8KB	STD (σ)	16KB	STD (σ)	32KB	STD (σ)
RAW (Mbps)	21,914	0,381	22,139	0,343	22,814	0,421	23,108	0,396	23,363	0,414	23,743	0,442
Virtual interface (Mbps)	21,047	0,511	21,243	0,547	21,984	0,650	22,455	0,682	23,022	0,715	23,482	0,788
Virtual Interface w/ Power saving mode on (Mbps)	21,001	0,523	21,109	0,581	21,807	0,694	22,393	0,702	22,900	0,763	23,374	0,802

Table 4.1: Wlan throughput in Mbps, different packet sizes.



Graph 4.1 Total throughput in Mbps, using one interface and one AP (TCP data).

The results we obtained for the first scenario show that the difference in terms of throughput between using or not the virtual interface for TCP, with just one interface, is very small, about 3.1% less in average with power saving off and 3.5% while on. Which is a good indicator, since in this topology the *vi* is simply relaying packets to the available interface. With more network interfaces and more available access points, this small delay added by the *vi,* can be easily

covered by the increase in throughput resulted from the simultaneous usage of those interfaces, as we will be seeing in experiment two and three.

Delay added by the *vi* module

Regarding the delay added by the *vi*, Table 4.2 shows the round-trip time in milliseconds and zero percentage of packet loss for the two same situations. This information allows us to have a very clear image of the delay added by the virtual interface since ping was used with the default packet size (64bytes). The smaller the packets, the more of them there are, which makes it easier to calculate the difference in terms of RTT between the control, designed as RAW, and our solution, since the differences are most likely caused by the *vi* module The values are taken during 100 seconds and averaged over 20 readings.

	Average RTT (ms)	**Std. Deviation (σ)**
RAW	1,730	0,009
Virtual Interface	1,799	0,016
Virtual Interface w/ Power saving mode on	1,808	0,014

Table 4.2: Ping results, 1 interface and 1 access point.

As we can see by the results presented in Table 4.2 the delay added by the *vi,* while the power saving mode is off, is around 4.0%, and roughly 4.2% with the power saving mode on, which is almost irrelevant (less than 0,1ms). The standard deviation also increases, while using the *vi*, since it is periodically estimating the RTT values for the available interfaces, which causes some fluctuations in terms of the total throughput. Also, there was zero percent packet loss for all three test situations.

The obtained results proves that using the *vi* with only one interface, does not causes too much impact in terms of total throughput or adds excessive overhead, which was a negative factor in previous versions.

Handover

Still in Experiment 1, we calculated the handover time, using the command *ping*. The number of missing packets were counted and multiplied by the ping frequency.

The type of handover that was measured was the horizontal handover, where the MAC level protocol remains the same, but the route changes. We trigger a route change by physically detaching the interface of a node. The results are presented in Table 4.3. Handover times are averages over 20 readings with standard deviation σ. Entries of the form "VI/x" must be understood as "Virtual Interface with a maxdiff value of x".

Type	Interface	Time (s)	Standard Deviation (σ)
Horizontal	RAW	1.5	0,5
	Vi/10	1.8	0.92
	Vi/100	2.3	0.73
	Vi/1000	2.5	0.67

Table 4.3: Handover time in seconds, using Wi-Fi interfaces.

Under our setup, when the *vi* was not being used, we measured a packet loss of roughly 1.5 packets when the route changed from one hop to another. Each packet that is lost corresponds to roughly one second passed by, since the ping frequency that was used was one second.

From the results presented in Table 4.3 we see that packet loss increases with increasing *maxdiff* threshold. The former is reasonable because the bigger the *maxdiff* value, the more the priority policy gets enforced, and a pure priority driven MAC switching would not lead to any switching at all. As expected, the smaller *maxdiff* gets the less stable the handover becomes. However, in our scenario a *maxdiff* value of 10 was sufficient to guarantee stable handover while changing interface priorities.

Power Consumption

Finally in Experiment 1, we calculated the amount of energy the mobile consumed during 600 seconds, with and without the *vi* module, so we could understand this abstraction impact on the energy being consumed by the mobile device. The values in Table 4.4 are presented in miliwatt hour (mWh). They were measured during 600s and averaged over 10 readings. To measure the consumed energy during this period we used the bash script presented in the Annex VI. It extracts the energy the device has, for each 10 seconds. The following table presents both the average energy consumption in mWh per second, and the total energy consumed during the 600s period.

	Average Energy Consumption (mWh) per second	Average Energy Consumption (mWh) in 600s	Standard Deviation (σ)
RAW	5,294	3123,333	1,035
Virtual Interface	5,316	3136,667	1,209

Table 4.4: Energy consumption in milliwatt hour.

As we can see in Table 4.4, the difference in terms of energy consumed during 600 seconds, between both scenarios, was only 13,33 mWh, just 0,427% more. This result proves that the module consumes just a minor added amount of energy when compared with a control scenario, designed as RAW in Table 4.4.

Overall there is a slight overhead when relying on the *vi*, be it from a throughput, power consumption, or from a delay perspective. This is expected, as by adding a layer of abstraction, we are also adding overhead in computation, with the expectations to introduce significant advantages.

4.2.2 Experiment 2

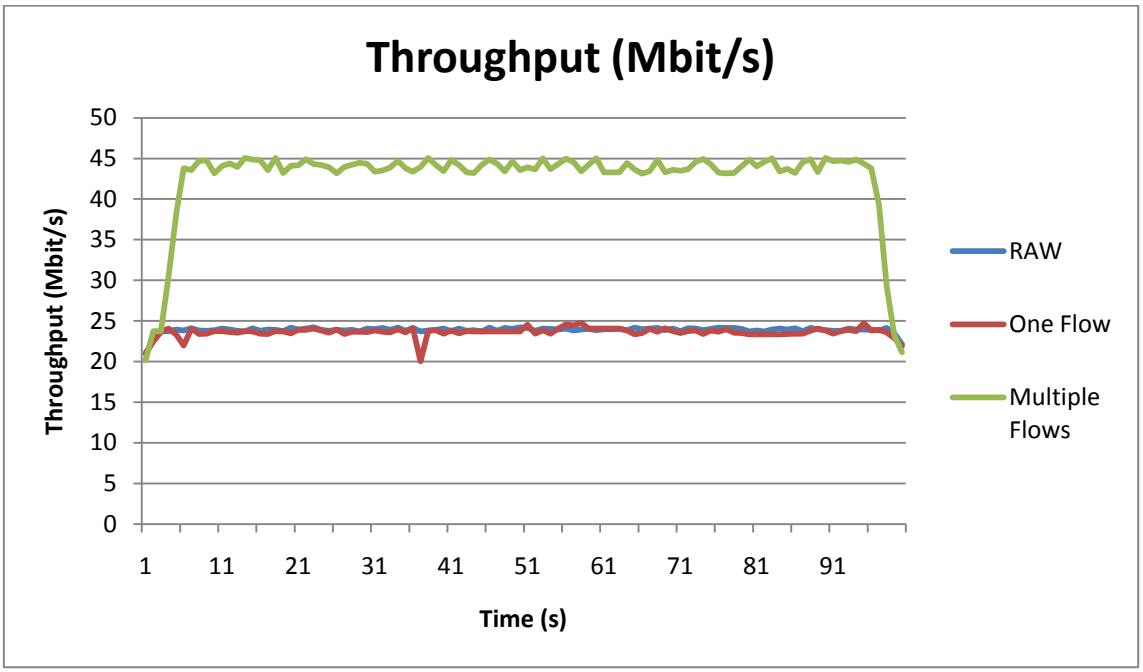
As in the previous experiment, node A is the single source transmitting data and node Dest. corresponds to the destination, but in this case we increased the number of interfaces present in the mobile device as well as the number of access points to which they are connected to. The experiment 2 is divided in two scenarios, both rely on Topology II (cf. section 4.1.2). While in the first test, the two existing APs are only being used by node A, in the second test, AP C is also being used by a second node that is constantly sending data, to simulate a saturated AP. The two physical interfaces used for this experiment were IEEE 802.11g.

In this experiment we investigate wherever the virtual interface is capable of detecting a saturated AP and reducing the amount of data a certain physical interface is sending to it, by diverting part of the traffic to a second physical interface that is using a less saturated AP.

Two Access Points, no saturation

For this scenario we used one data flow coming from a single application and multiple data flows coming from different applications, so the available interfaces could be used simultaneously. Starting with the analysis of the total throughput using TCP data, Graph 4.2

shows the throughput when using the virtual interface with one and several different data flows during 100 seconds, averaged over 20 readings. The packet size used for this experiment was 32 Kbytes.



Graph 4.2: Total throughput in Mbps, using the *vi* with two interfaces and two APs (TCP data).

	Average Throughput (Mbit/s)	Standard Deviation (σ)	Packet loss (%)
One Data Flow	23,627	0,610	1%
Multiple Data Flows	42,615	5,161	3%
RAW	23,879	0.410	0%

Table 4.5: Average throughput in Mbps, using the *vi* with two interfaces and two APs.

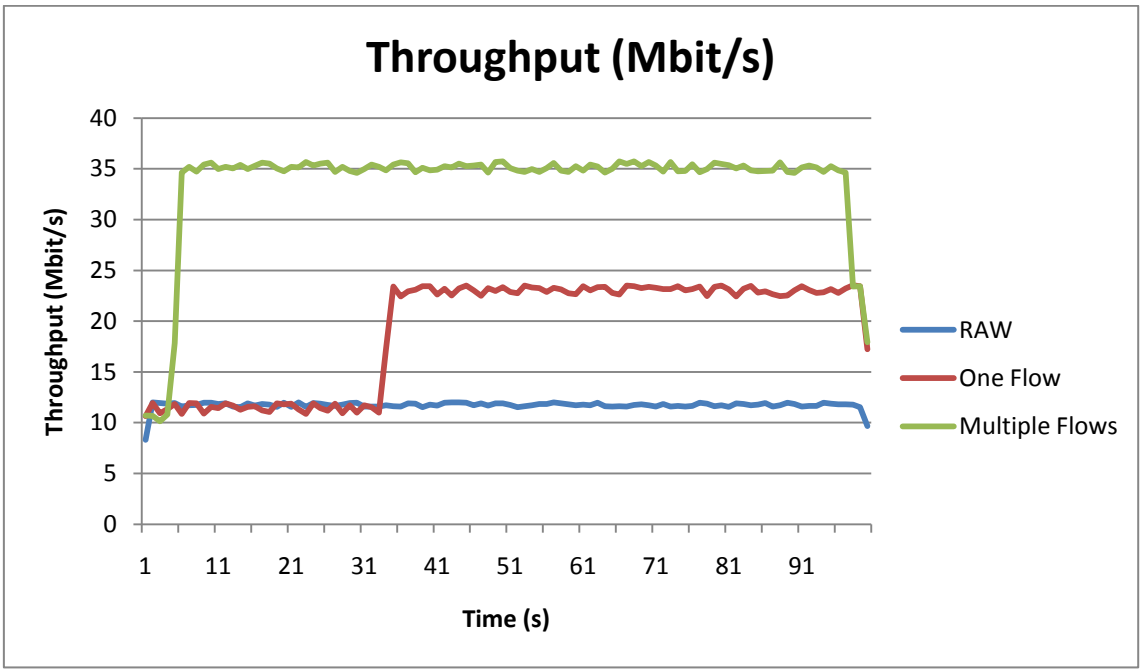
As we can see in the results presented by Graph 4.2, the total throughput when using the virtual interface with TCP data (while sending a single data flow) is very similar in comparison with *RAW* measurements taken without the virtual interface. On the other hand the values obtained when sending different data flows, are nearly 79% higher when comparing with the RAW measurements from Experiment 1. This increment results from the fact that we are simultaneously using several physical interfaces to send the different data flows.

Also if we look closely in Graph 4.2, it takes around 5 seconds for the throughput to reach more than 40Mbit/s, this happens because the mobile device will only be using both physical interfaces simultaneously, when the first interface finishes transmitting the data from the first data flow in its queue and moves on to the second data flow. This is done to ensure that we will not have a huge amount of packets reaching the correct destination out of order.

The packet loss is in average 1% and 3% (cf. Table 4.5) for one data flow and multiple data flows respectively, due to the fact that we are constantly switching the physical interfaces used to transmit the data, which causes some packets to be lost and some fluctuations in terms of throughput, raising also the standard deviation.

Two Access Points, AP C is saturated

For this scenario we saturated one of the access points, to verify how would the *vi* adapt to a sudden increase in terms of the RTT value measured for a certain physical interface, connected to that AP. As in the previous test experiment, single and multiple data flows were employed. The values obtained were taken during a time frame of 100 seconds, and averaged over 20 readings. For a better comparison, we also measured the throughput of a single interface, without using the *vi* module, designated as *RAW*, connected to a single saturated AP. The packet size used for this experiment was 32 Kbytes.



Graph 4.3: Total throughput in Mbps, with two interfaces and two APs (one saturated) (TCP data).

For one continuous data flow, the total throughput when using the vi, during the time frame 1 to 30 seconds is very similar to the RAW measurements (cf. Graph 4.3), since the *vi* is actually using the interface connected to the saturated AP. Once it changes to the second AP that is currently not being used by another user, it causes a sudden increase in the total throughput, reaching roughly 24 Mbps as seen in Graph 4.3. What happens is that the VBA assigns a lower percentage to the interface using the saturated AP, which will lower the time that such interface will be used to send the data, increasing the total throughput to an average of 19,125 Mbps as seen in Table 4.5.

In this specific scenario, with a single data flow, the usage of the *vi* module with two different physical interfaces is in average, increasing 63% the total throughput, when comparing with a situation where a single interface is connected to a saturated AP (11,713 Mbps).

When sending different data flows, we obtained a very similar result to the one presented in the previous scenario (2 APs, no saturation). It takes around 6s for the throughput to reach 35Mbps, when the *vi* starts using both physical interfaces, and consequently the two APs. Since one of the APs is dividing the bandwidth between node A (cf. Figure 4.2) and a third party user, the total throughput when using both network interfaces only reaches a little more than 35Mbps in comparison with the 42Mbps obtained in the previous test scenario.

	Average Throughput (Mbit/s)	**Standard Deviation (σ)**	**Packet loss (%)**
One Data Flow (vi)	19,125	5,491	2%
Multiple Data Flows (vi)	33,575	5,551	3%
RAW	11,713	0,428	1%

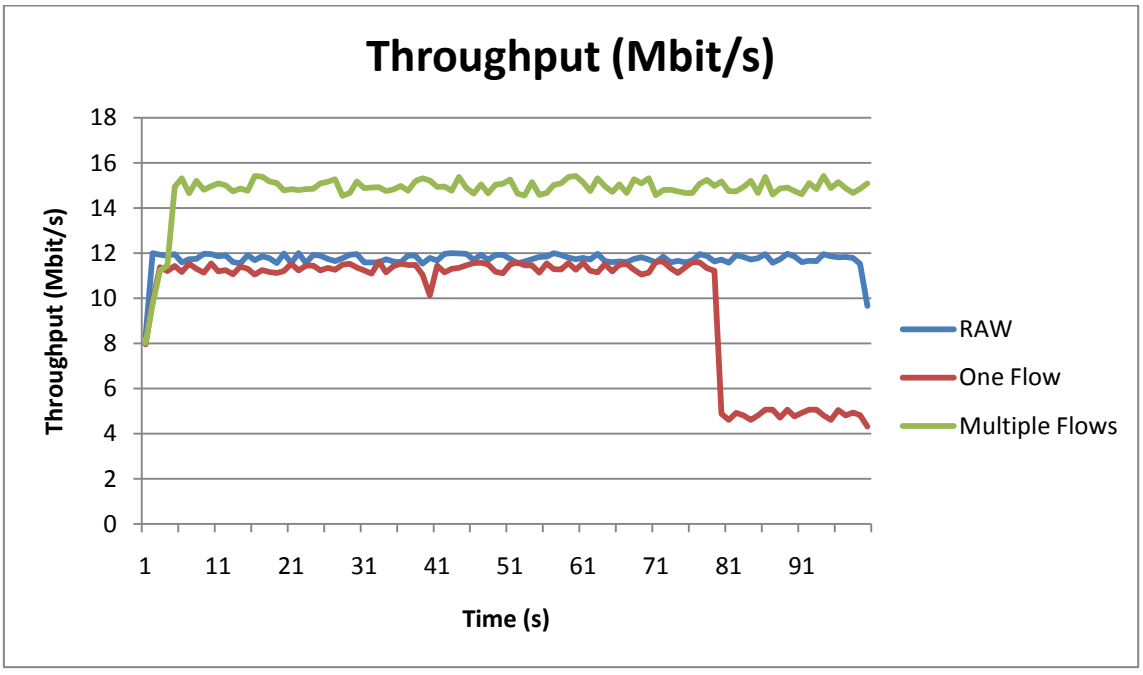
Table 4.6: Throughput in Mbps, using two interfaces and two APs (one saturated).

The load balancing mechanism presented in the *vi* module slightly increases the packet loss and standard deviation as seen in Table 4.6. If we compare both the standard deviation results of this and the previous scenario, when sending a single and continuous data flow, there is a relatively high increment, which is caused by the difference in throughput values obtained for the two access points (11.5 Mbps and 23.5 Mbps).

4.2.3 Experiment 3

To understand and find the potential limitations of the *vi* module, we decided to create a worst-case scenario. It relies on Topology III (cf. section 4.1.2), where we have three physical network interfaces (two IEEE 802.11g and one IEEE 802.11b) connected to just one saturated access points. For this experiment we also used one single and continuous data flow coming from a single application, and multiple data flows as well, so that the available interfaces could be used simultaneously.

Starting with the analysis of the total throughput using TCP data, Graph 4.4 shows the throughput when using the virtual interface with one and several different data flows during 100 seconds, averaged over 20 readings. The packet size used for this experiment was 32 Kbytes. In Graph 4.4 we also added the throughput of a single interface, calculated without using the *vi* module, designated as *RAW*, connected to a single saturated AP (values calculated in experiment 2).



Graph 4.4: Total throughput in Mbps, with three network interfaces, and one saturated AP (TCP data).

If we take in consideration that this is a worst case scenario, where all access points are being used by several users, the results presented in Graph 4.4, are very satisfactory. While sending a single and continuous data flow the throughput is slightly lower when comparing with the RAW measurements, due to the fact that there is a network interface with a lower bandwidth

(IEEE 802.11b) than the other two. This network interface is used during a smaller period of time, meaning the load balancing mechanism detected that the estimated RTT for that link was higher than the remaining.

When using multiple data flows the total throughput increases due to the fact, that multiple network interfaces are used simultaneously. In this case, by establishing multiple connections to a single AP, its available interface is distributed equally through all active users.

When testing with just one network interface, without the *vi* module, we will have two active users connected to the available AP, our physical interface and third party user saturating the AP, so its bandwidth will be divided between the two users. For this experiment when using the *vi* module, we had three physical interfaces plus the third party user connected to the same AP, which means we are getting roughly three quarters of the AP's bandwidth instead of just half, resulting in a increment of the total throughput. Therefore, the percentage of total throughput is a function of the ratio of interfaces (instead of ratio of users).

	Average Throughput (Mbit/s)	**Standard Deviation (σ)**	**Packet loss (%)**
One Data Flow (vi)	9,915	2,664	2%
Multiple Data Flows (vi)	14,756	1,021	4%
RAW	11,713	0,428	1%

Table 4.7: Throughput in Mbps, using three interfaces and one saturated AP.

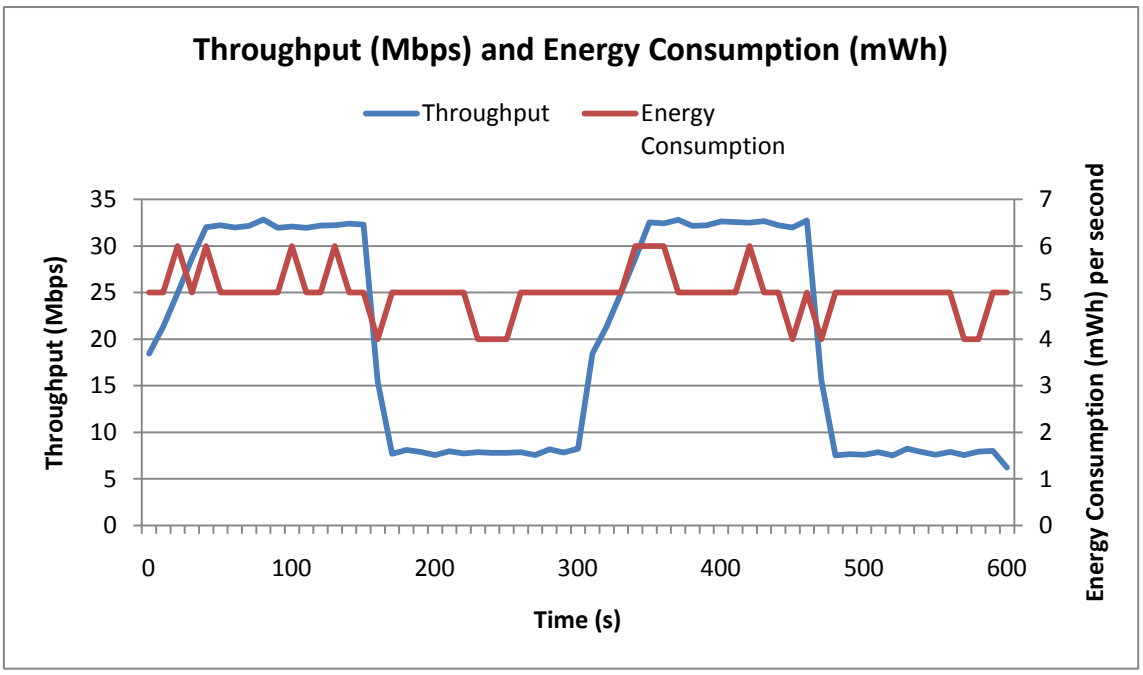
As seen in Table 4.7 both the packet loss and standard deviation, for the two test experiments are above the values obtained for the control scenario, described as RAW. This difference can be easily explained due to the fact that for this experiment we have three network interfaces, one of them has a lower bandwidth than the other two, which causes a bigger fluctuation in terms of total throughput. The one percent increment in the packet loss may result from the usage of multiple interfaces in a saturated environment.

4.2.4 Experiment 4

Finally in Experiment 4 we tested the power consumption mechanism, to try to understand if by using fewer interfaces to send the data we are able to diminish the amount of energy being spent and if the difference is significant. This experiment relies on Topology II (cf. section 4.1.2), where node A is the single source transmitting data and node Dest. corresponds to the

destination. The two network interfaces (1x IEEE 802.11g, 1x IEEE 802.11b) are connected to two access points. Real traffic (different data flows, from different applications) was used, to simulate a real-life environment.

We used the bash script presented in Annex VI to extract the amount of energy being consumed every 10 seconds, and *jperf* to measure the throughput. The data for both measurements was collected over 600 seconds (10 minutes) and is presented in Graph 4.5. Each measurement is displayed according to its own axis, so we can then verify if there is a correlation between the throughput and the energy values. The energy consumption is presented in miliwatt hour (mWh), while the throughput is displayed in Mbps.



Graph 4.5: Correlation between the total Throughput in Mbps and the Consumed Energy in miliwatt hour.

We can clearly verify from Graph 4.5, that when there is an increment is terms of throughput, there is also a slight increment is terms of the energy being consumed by the device. The opposite is also true. This happens because the power consumption mechanism, depending on the size of the data flow, decides if the device should use the network interfaces picked by the VBA or the interface with the lowest energy consumption to send the data.

When a data flow has a significant size, it is usually better to maintain the VBA's policy, since the device will transmit the data at a higher rate, which will result in a lower amount of energy being consumed to send that amount of data. When data flows have a relatively small size, it takes roughly the same time to send them when using one or several network interfaces,

resulting in a noticeable energy saving when using just one of those interfaces. This is exactly what we see in Graph 4.5.

During this experiment, the mechanism saved roughly 190 mWh of battery in our device, which for a laptop is not very significant, but in a smaller and more economic device, such as a mobile phone or a sensor, this difference can have a very big impact, since it gives extra time of battery.

4.3 Performance Evaluation Summary

The results we obtained in terms of delay added by the *vi* module are very promising, especially if we compare them with the *vi*'s previous version results [28][32], where the added overhead was so high that it caused the throughput to go down around 30%, mostly caused by the way the authors used to intercept the data. Another expected result was the packet loss percentage getting higher with the usage of several interfaces and APs, which is normal, since the number of available paths to send the information also increases.

By adding the aggregation and load-balancing mechanisms to the virtual interface we were able to significantly increase the total throughput in around 70% (average) when sending multiple data flows, by using multiple network interfaces simultaneously, proving that it is possible to have a virtual interface hiding the heterogeneity of the used devices from the upper layers, without adding an excessive amount of delay, and increasing the total throughput, even in a worst-case-scenario.

In terms of the power consumption mechanism, the obtained values are also very promising; by using this mechanism we were able to optimize the amount of energy being consumed. The amount of saved energy is relatively low, but for a small device, such as a mobile phone or a sensor, with very strict energy concerns, such difference can be quite significant, resulting in extra time of battery.

We are now able to answer the questions presented in the beginning of this chapter:

Is the overhead added by the virtual interface excessive?

By adding Netfilter hooks we were able to significantly reduce the amount of delay added by the *vi* module, which is now less than 0,1ms, resulting in a total increment of 4%. The obtained results prove that using the *vi* with only one network interfaces, does not causes a high impact

in terms of total throughput nor does it add excessive overhead, which was a negative factor in previous versions.

Are the implemented mechanisms improving the total throughput? If yes, in which situations?

In all experiments, when using multiple data flows, we were able to significantly increase the total throughput when using the *vi* module with multiple interfaces, even in a worst-case-scenario with multiple network interfaces and just one available AP. When sending just one continuous data flow, the average throughput is slightly lower when there is no saturation of the APs being used, but it is higher when at least one AP is saturated, since the load balancing mechanism is able to detect the saturated link, and reduce the amount of data being sent via that network interface.

Overall, using this type of virtual interfacing is extremely valuable for multiple simultaneous flows (e.g. use of multiple applications) which in fact corresponds to today's majority of situations. Assuming a single flow, then the added cost is not significant, comparing to the benefits provided overall for multiple flow usage.

Is the power consumption mechanism, present in the *vi* module, saving any energy?

By using the *vi* module, with the power consumption mechanism on, we were able to reduce the amount of energy being consumed by using a network interface with a lower energy consumption. The amount of energy saved is not very high but if we apply the solution to devices with very strict power limitations, such as mobile phones, such result will be beneficial. In our device, the usage of this power consumption mechanism resulted in energy saving of roughly 5.4%.

5 Conclusions and Future Work

This chapter relates to the summary and conclusions to be drawn from the work developed.

We implemented an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc networks. Such networks are characterized by different MAC technologies used among the nodes. The solution is based on a *virtual interface* (vi) approach, which allows the usage of all interfaces presented in a mobile device simultaneously, while hiding the heterogeneity from the applications and allow any number of interfaces to be added, increasing the total throughput.

Since we are looking for a virtual interface, we are not interested in packet forwarding, but the idea of storing a MAC/interface mapping based on incoming packets is also suitable for local traffic. We have implemented a virtual interface (*vi*) that adopts this mechanism. Like the Linux Ethernet Bridge, the *vi* represents a regular layer-two-device and can be configured accordingly.

Using a custom Netfilter target has shown to be very promising solution, that added very little overhead and proved to be very flexible. Such a target can be loaded and unloaded from kernel at any time.

Implementing a virtual interface for transparent heterogeneous mobile ad-hoc networks has proven to be a good approach. Reasonable handover times can be achieved when using any routing protocols. The throughput rates when using the *vi* module, with several network interfaces, are significantly higher, reaching in some situations an increase of 79%. In terms of the power consumption mechanism, the experimental values are also very promising; by using this mechanism we were able to optimize the amount of energy being consumed.

As future work, we would like to extend the virtual interface to work in different access networks, others than ad-hoc networks, with for example several 3G and 802.11x interfaces. Moreover, it would be interesting to extend the module to smaller devices, such as mobile phones, to see how it would impair their performance in terms of their total throughput and energy consumption.

Finally, in order to simplify the process of estimating the energy being consumed for every network interface, we considered that the energy consumption of a certain interface when in idle time is zero. For a more correct approach, the energy a certain interface is consuming when in idle, should also be taken as a variable in the used algorithm. It would be interesting to make such modification and compare the results, so we could understand if there is actually any impact in the amount of energy saved by the power saving mechanism, when considering the idle time as an additional variable.

6 Bibliography

[1] S. Dhawan, “Analogy of Promising Wireless Technologies on Different Frequencies: Bluetooth, WiFi, and WiMAX”. In *The 2nd International Conference on Wireless Broadband and Ultra Wideband Communications (AusWireless 2007)*, Sydney, August 2007.

[2] L. Feeney, B. Ahlgren, and A. Westerlund, “Spontaneous networking: an application-oriented approach to ad-hoc networking”, *IEEE Communications Magazine, 39(6)*, June 2001.

[3] L. Chen, S. Das, M. Gerla and A. Nandan, “Moving between infrastructure and ad-hoc wireless networks: 'opportunistic' mobile middleware”, Computer Science Department, UCLA, USA, 2005.

[4] L. Carvalho, R. Sofia. “User-provided Networks: Relaying vs. Ad-hoc Routing”, MSc thesis, Universidade Porto/INESC Porto, 2009.

[5] C. E. Perkins, E. M. Belding-Royer, and S. Das. “Ad-hoc On-Demand Distance Vector (AODV) Routing”. IETF Standard *RFC 3561*, July 2003.

[6] C. E. Perkins and E. M. Royer. “The Ad-hoc On-Demand Distance Vector Protocol”. In *C. E. Perkins, editor, Ad-hoc Networking, pages 173–219*. Addison-Wesley, 2000.

[7] Uppsala University CoRe Group. “Aodv-uu”. Dec. 2007 [Online]. Available: http://core.it.uu.se/core/index.php/AODV-UU. [Accessed: Aug. 2010].

[8] H. Lundgren, E. Nordstr¨ m, and C. Tschudin. “Coping with Communication Gray Zones”. In *IEEE 802.11b based Ad-hoc Networks. Technical Report 2002-022*, Uppsala University Department of Information Technology, June 2002.

[9] OLSRD Project. Olsr daemon. [Online] Available: http://www.olsr.org. [Accessed: Aug. 2010].

[10] Chandrasekhar and J. Andrews, “Femtocell Networks: A Survey”, *IEEE Communications Magazine*, Vol. 46, no. 9, pp 59 – 67, 2008.

[11] C. E. Palazzi, G. Pau, M. Roccetti, M. Gerla, “In-Home Online Entertainment: Analyzing the Impact of the Wireless MAC-Transport Protocols Interference”, *IEEE WIRELESSCOM2005*, USA, June 2005.

[12] C. E. Palazzi, S. Ferretti, M. Roccetti, G. Pau, M. Gerla, “What's in that Magic Box? The Home Entertainment Center's Special Protocol Potion, Revealed”, *IEEE Transactions on Consumer Electronics*, vol. 52, no. 4, pp. 1280-1288, November 2006.

[13] C. E. Palazzi, N. Stievano, M. Roccetti, “A Smart Access Point Solution for Heterogeneous Flows”, Universitá di Padova, Italy 2009.

[14] S. Charoenpanyasak and B. Paillassa, “SCTP multihoming with Cross Layer Interface in Ad-hoc Multihomed Networks”, *Third IEEE International Conference on Wireless and Mobile Computing*, Networking and Communications, France 2007.

[15] Y. Choi, B. Kim, S. Kim, M. In, and S. Lee, “A Multihoming Mechanism to Support Network Mobility in Next Generation Networks”, IEEE 2006.

[16] Liu and A. Goldsmith, “Load-balancing and Switch Scheduling”, IEEE 2005.

[17] N. Spring, R. Mahajan and D. Wetherall, "Measuring ISP topologies with rocketfuel". In *Proceedings ACM SIGCOMM*, Philadelphia, August 2002.

[18] Habib, N. Christin and J. Chuang, “On the Feasibility of Switching ISPs in Residential Multihoming”, Carnegie Mellon University, California, 2007.

[19] T. V. Eicken and W. Vogels, “Evolution of the Virtual Interface Architecture”, USA, 1998.

[20] Z. Li, L. Fu, Ke Xu, Z. Shi and J. Wu, “Smooth Handoff Based on Virtual Interface in Wireless Network”. In *Wireless and Mobile Communications, ICWMC '06. International Conference*, Bucharest, July 2006.

[21] C. Tsao and R. Sivakumar, “On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts”, Georgia Institute of Technology, December 2009.

[22] Netfilter - Firewalling, NAT and packet mangling for Linux [Online] Available: http://www.netfilter.org/ [Accessed: Aug. 2010].

[23] H.-Y. Hsieh and R. Sivakumar. “A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts”. In *MobiCom '02 Proceedings of the 8th annual international conference on Mobile computing and networking*, Atlanta, September 2002.

[24] K.-H. Kim, Y. Zhu, R. Sivakumar, and H.Y. Hsieh, “A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces”. *Wireless Networks*, 2005.

[25] L. Magalhaes and R. Kravets, “Transport level mechanisms for bandwidth aggregation on mobile hosts”. In *Proceedings of INCP*, November 2001.

[26] Bluetooth, Bluetooth Network Encapsulation Protocol (BNEP) Specification, June 2001.

[27] Media Access Control (MAC) Bridges, June 2004,

[28] P. Stuedi and G. Alonso, “Transparent Heterogeneous Mobile Ad-Hoc Networks”. In *Proceedings of the Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous’05)*, San Diego, USA, July 2005.

[29] James T. Yu, “Performance Evaluation of Linux Bridge”, DePaul University, 2004.

[30] IEEE Computer Society. IEEE 802.11 Standard, IEEE Standard For Information Technology, 1999.

[31] The Netfilter project team, “Linux Netfilter/Iptables frameworks”, Nov 1999. [Online]. Available: http://www.netfilter.org/. [Accessed: Aug. 2010].

[32] S. Graf, “Implementing a virtual network interface for heterogeneous mobile ad-hoc networks (802.11 and Bluetooth)”, Swiss Federal Institute of Technology, Zurich, August 2006.

[33] J. Mota, A. Arsénio and R. Sofia, “Combining Heterogeneous Access Networks with Ad-Hoc Networks for Cost-Effective Connectivity”. In *API Review 2010, 1º volume, Edições Lusófonas*, February 2011.

[34] Linux Diagnostic Tools – System Utilities based on Sysfs. [Online] Available: http://linux-diag.sourceforge.net/Sysfsutils.html. [Accessed: June 2010].

[35] Mirzaie, S. Elyato, A.K. Sarram, M.A., “Preventing of SYN Flood Attack with Iptables Firewall”. In *Communication Software and Networks, 2010. ICCSN '10. Second International Conference,* Singapore, February 2010.

[36] P. Karn and C. Partridge. “Improving round-trip time estimates in reliable transport protocols”. In *Proceedings of the SIGCOMM ’87 Conference*, Stowe, Vermont, August 1987.

[37] V. Jacobson. “Congestion avoidance and control”. In *Proceedings of the SIGCOMM ’88 Conference*, Stanford, California, August 1988.

[38] J. Cano and P. Manzoni. “A Performance Comparison of Energy Consumption for Mobile Ad Hoc Network Routing Protocols”, IEEE. Valencia, 2000.

[39] Alessandro Corbet and Greg Kroah-Hartman. “Linux Device Drivers”. O’Reilly Media, 3rd edition, 2005.

[40] Cross-Referencing Linux. Lxr. [Online] Available: http://lxr.linux.no. [Accessed: Aug. 2010].

[41] Free Software Foundation. Gnu general public license. [Online] Available: http://www.gnu.org/licenses/gpl.html. [Accessed: Aug. 2010].

[42] Linux Diagnostic Tools. sysfsutils. [Online] Available: http://linux-diag.sourceforge.net/Sysfsutils.html [Accessed: Aug. 2010].

[43] Uppsala University CoRe Group. Aodv-uu. [Online] Available: http://core.it.uu.se/AdHoc/AodvUUImpl. [Accessed: Aug. 2010].

[44] IEEE Std. 802.11, “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” 1999.

[45] IEEE Std. 802.11e, “Wireless LAN medium access control (MAC) and Physical Layer (PHY) Specifications: medium access control (MAC) enhancement for quality of service (QoS),” 2002.

[46] IEEE Std. 802.11b, “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-speed Physical Layer Extension in the 2.4 GHz Band,” 2001.

[47] IEEE Std. 802.11g, “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Further Higher-Speed Physical Layer Extension in the 2.4 GHz Band,” 2003.

[48] K. Medepalli, “Voice capacity of IEEE 802.11b, 802.11a, and 802.11g wireless. LANs”. In *Proc. IEEE Globecom*, pp.1549–1553, 2004.

[49] Hewlett-Packard, Understanding Bluetooth™, January 2002.

Annex I – Wi-Fi and Bluetooth

IEEE 802.11 – Wi-Fi

Since its entrance into the mainstream of networking technology, Wi-Fi has mostly been used as a replacement and augmentation for wired local area networks. Wi-Fi is well-suited for applications requiring high-volume data transfer and distances below 10 meters.

WLANs have been, through the last decade, deployed as an extension of other access technologies in a way to expand the reach of Internet broadband access and hence to facilitate the penetration of *Voice over IP* (VoIP) and other data services. WLANs as complementary networks normally follow an infrastructure mode of operation, where a central controller - the Access Point (AP) / meddles and controls communication between any 2 elements.

More recently, there has been a surge of WLANs operating in mesh (ad-hoc mode), that is, in a completely decentralized way. This is due to the emergence of soft-radio and also of open distribution operating systems contemplating low-cost APs.

In both situations, the widespread deployment of WLANs is underpinned by the two most popular variants of IEEE 802.11 standards, 802.11b and 802.11g.

IEEE has specified a set of standards as the 802.11 family for WLANs. The IEEE 802.11 specifications define a single *Medium Access Control* (MAC) layer [44][45] along with multiple physical layers [44][46][47]. *Distributed Coordination Function* (DCF) is the fundamental MAC technique that employs a *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) distributed algorithm and an optional virtual carrier sense using *Request To Send* (RTS) and *Clear To Send* (*CTS*) control frames. The original IEEE 802.11 standard [44] specifies data rates of 1 Mb/s and 2 Mb/s, and defines *Direct Sequence Spread Spectrum* (DSSS) - based physical layer that operates at the 2.4 GHz ISM band. The original 802.11 was rapidly supplemented by IEEE 802.11b [46], which is specified to support higher data rates up to 11 Mb/s at 2.4 GHz using DSSS with complementary code keying (CCK) modulation.

IEEE 802.11g further extends 802.11b to support the data rates up to 54Mb/s at 2.4GHz. The higher data rate in 802.11g is enabled by using *Orthogonal Frequency Division Multiplexing* (OFDM) modulation as specified in the so-called *Extended Rate Physicals* (ERPs) physical layer.

Along with the gradual deployment of WLANs, the involved Wi-Fi products are delivered under different versions of 802.11 standards. Thus, IEEE 802.11b and IEEE 802.11g devices unavoidably co-exist in common coverage area. In this mixed networking environment, the legacy 802.11b devices cannot detect the ERP-OFDM signals sent from 802.11g devices;

consequently they cannot cause the *Clear Channel Assessment* (CCA) function within physical layer to indicate the channel busy and refrain from channel access as specified in CSMA/CA. This inability of legacy 802.11b devices leads to frame collisions in the channel access between 802.11b and 802.11g devices. To deal with this issue, the 802.11g defines a protection mechanism based on the channel reservation for the ERP-OFDM transmissions. To ensure that the reserved channel status is understandable by the mixed devices, extra frames are introduced in the protection mechanism. Those frames have to be sent with NonERP modulation at a low rate (typically 2 Mb/s) for them to be understood by all stations.

One option of extra frames is the RTS/CTS frames which are originally designed to reduce frame collisions caused by hidden terminals. Any device (other than sender and intended receiver) receiving the RTS or CTS frames should refrain from sending data by setting its *network allocation vector* (NAV) for a given time period indicated in the *Duration* field of RTS and CTS frames. The other option of extra frames is CTS-to-elf frames whose source address and destination address are identical. The 802.11g sender transmits a CTS-to-self frame to inform all the neighbouring 802.11g and 802.11b devices to update NAV according to the *Duration* field of the CTS-to-self frame. Obviously, extra frames (RTS, CTS, and CTS-to-self) used for ensuring interoperability are viewed as overhead for system performance because they reduce the available medium resource for data delivery.

Specifically, when voice traffic is provisioned over homogeneous 802.11 WLANs (802.11b only WLAN or 802.11g only WLAN), exchange of RTS and CTS frames is typically turned off because a VoIP packet size (200 bytes in G.711) is usually less than a pre-defined triggering threshold (maximum is 2347). In the mixed 802.11b and 802.11g WLAN, either exchange of RTS/CTS frames or sending CTS-to-self frames needs to be initiated for performing the protection mechanism.

CTS-to-self protection mechanism is more efficient than RTS/CTS protection mechanism in clear channel conditions (no hidden terminals). Usually, there are few hidden terminals in the indoor voice over WLAN (VoWLAN) services, hence CTS-to-self protection mechanism is typically utilized [48].

Nevertheless, the voice performance degrades significantly in the mixed 802.11b and 802.11g WLAN with either protection mechanism. Compared to the 802.11g only WLAN, it is reported in [47] that the voice capacity in the mixed WLAN drops more than 70% and 50% with RTS/CTS protection mechanism and CTS-to-self protection mechanism, respectively.

IEEE 802.15.x – Bluetooth

Bluetooth is defined as a wireless technology that provides short-range communications intended to replace the cables connecting portable and/or fixed devices while maintaining high levels of security. There are three key features of Bluetooth; robustness, low power, and low cost. The Bluetooth standard provides a uniform structure enabling a wide variety of devices to seamlessly, and wirelessly, connect and communication with each other. Bluetooth devices connect and communicate via RF link through short-range piconets and have the ability to connect with up to seven devices per piconet. Each of these devices can also be simultaneously connected to other piconets.

The piconet itself is established dynamically and automatically as Bluetooth enables devices enter and leave the range in which its radio operates. The major pro of Bluetooth is the ability to be full duplex and handle both data and voice transmission simultaneously. The differentiation of Bluetooth from other wireless standards such as Wi-fi is that the Bluetooth standard gives both link layer and application layer definitions which support data and voice applications.

Bluetooth comes in two core versions; Version 2.0 + Enhanced Data Rate and Version 1.2. The primary differences being Bluetooth 2.0 has a data rate of 3 Mega bits per second whereas Version 1.2 has only a 1 Mega bit per second data rate. Both are equipped with *extended Synchronous Connections (eSCO)*, which improves voice quality of audio links by allowing retransmissions of corrupted packets.

Bluetooth technology operates in the unlicensed industrial, scientific and medical (ISM) band at 2.4 to 2.485 GHz, using a spread spectrum, frequency hopping, full-duplex signal at a nominal rate of 1600 hops/sec. Bluetooth is modulated using adaptive frequency hopping (AFH). This modulation has the capability to reduce interference between wireless technologies sharing the ISM band. It does this by having the ability to detect other devices using the ISM band and use only frequencies that are free. The signal itself hops between ranges of 79 frequencies at 1 Megahertz intervals to minimize interference [49].

The devices themselves are categorized into range ability. There are three classes of devices each covering a select range. Class 1 devices are mostly used in industrial cases and have a range of 100 to 300 meters. These devices take more power than the standard devices you and I are accustomed to in our daily routine and therefore are a bit more expensive. Class 2 devices are most commonly found in mobile devices and the most commonly used. Items such as cell phones and printers are Class 2 devices and have a range of 10 to 30 feet and use only 2.5 milli-Watts of power. Finally, Class 3 devices have the shortest range of up to 1 meter and include devices such as keyboards and a computer mouse. Class three devices therefore require the least amount of power and are in general the lease expensive.

The Bluetooth specification defines two links types, *Asynchronous Connectionless* (ACL) and *Synchronous Connection Oriented* (SCO). Different link types can be used by different master slave pairs in the same piconet. The SCO links are chiefly used for voice traffic and their data rate is 64 Kbps. These are characterized by a periodic single slot packet assignment. For data traffic and support broadcast messages ACL links are mainly used. ACL link types are used by Multislot packets and can attain maximum data rate of 721 Kbps in one direction and 57.6 Kbps in other direction. These data rates can be achieved if no error correction is used.

For device communication the Bluetooth specification uses Time Division Duplexing and Time Division Multiple Access, being 625 μ sec the length of single time slot [49].

There is a preset packet format for Bluetooth. Firstly a 72 bit access code that holds the piconet address. The 54 bit header following the access code contains retransmission, flow control and error correction information. The payload field comes in the last of packet and may be of up to 2745 bits.

Annex II – AODV and OLSR

AODV - Ad-hoc On demand Distance Vector routing protocol

The *Ad-hoc On demand Distance Vector* (AODV) [5][6] is a routing algorithm for MANET, so that routes between nodes are only built as soon as, and maintained as long as, they are needed by a source node. Figure 6.1 shows the message exchanges of the AODV protocol.

Hello messages may be used to detect and monitor links to neighbors. If Hello messages are used, each active node periodically broadcasts a Hello message that all its neighbors receive. Because nodes periodically send Hello messages, if a node fails to receive several Hello messages from a neighbor, a link break is detected.

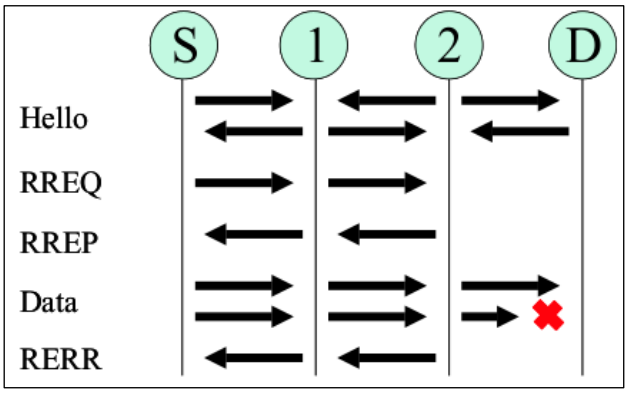


Figure 6.1: AODV protocol messaging.

When a source has data to transmit to an unknown destination, it broadcasts a *Route Request* (RREQ) for that destination. At each intermediate node, when a RREQ is received a route to the source is created. If the receiving node has not received this RREQ before, and it is not the destination and does not have a current route to the destination, it re-broadcasts the RREQ. If the receiving node is the destination or has a current route to the destination, it generates a *Route Reply* (RREP). The RREP is unicast in a hop-by- hop fashion to the source. As the RREP propagates, each intermediate node creates a route to the destination. When the source receives the RREP, it records the route to the destination and can begin sending data. If multiple RREPs are received by the source, the route with the shortest hop count is chosen.

As data flows from the source to the destination, each node along the route updates the timers associated with the routes to the source and destination, maintaining the routes in the routing table. If a route is not used for some period of time, a node cannot be sure whether the route is still valid; consequently, the node removes the route from its routing table.

If data is flowing and a link break is detected, a *Route Error* (RERR) is sent to the source of the data in a hop-by- hop fashion. As the RERR propagates towards the source, each intermediate node invalidates routes to any unreachable destinations. When the source of the data receives the RERR, it invalidates the route and re-initiates route discovery if necessary.

An implementation of AODV for Linux systems named AODV-UU is provided by the Uppsala University [7]. It consists of a kernel module and a userspace daemon. Although still experimental, AODV performs well in a Linux MANET.

AODV-UU uses *Netfilter* to capture the packets. The main protocol logic resides in a userspace daemon. The authors have also added a number of supplemental features, not part of the AODV draft, to increase the performance of Hello messages [8] (e.g., unidirectional link support and a signal quality threshold for received packets). In addition, AODV-UU also includes Internet gatewaying and multiple interface support. Since AODV-UU is well documented and able to run in simulation, a number of patches are available (e.g., multicast and subnetting) to further extend its functionality.

OLSR - Optimized Link State Routing protocol

The Optimized Link State Routing protocol is a proactive, table-driven routing algorithm for MANET. An implementation of the OLSR protocol is provided by [9]. OLSR runs as a standalone server process and is platform independent. It is supposed to work on Linux, FreeBSD, NetBSD, OS X and even on Windows. All operations are performed in userspace. Explicit interaction with the kernel is only necessary to manipulate the routing table.

OLSR minimizes the overhead from flooding of control traffic by using only selected nodes, called “multipoint relays” (MPRs), to retransmit control messages. This technique significantly reduces the number of retransmissions required to flood a message to all nodes in the network. Secondly, OLSR requires only partial link state to be flooded in order to provide shortest path routes. The minimal set of link state information required is, that all nodes, selected as MPRs, must declare the links to their MPR selectors. Additional topological information, if present, may be utilized e.g., for redundancy purposes.

OLSR may optimize the reactivity to topological changes by reducing the maximum time interval for periodic control message transmission. Furthermore, as OLSR continuously maintains routes to all destinations in the network, the protocol is beneficial for traffic patterns where a large subset of nodes are communicating with another large subset of nodes, and where the [source, destination] pairs are changing over time. The protocol is particularly suited for large and dense networks, as the optimization done using MPRs works well in this context. The larger and more dense a network, the more optimization can be achieved as compared to

the classic link state algorithm.

OLSR is designed to work in a completely distributed manner and does not depend on any central entity. The protocol does not require reliable transmission of control messages: each node sends control messages periodically, and can therefore sustain a reasonable loss for a number of such messages. Such losses occur frequently in radio networks due to collisions or other transmission problems.

Also, OLSR does not require the sequencial delivery of messages. Each control message contains a sequence number which is incremented for each message. Thus the recipient of a control message can, if required, easily identify which information is more recent, even if messages have been re-ordered while in transmission. Furthermore, OLSR provides support for protocol extensions such as sleep mode operation, multicast-routing etc. Such extensions may be introduced as additions to the protocol without breaking backwards compatibility with earlier versions. OLSR does not require any changes to the format of IP packets. Thus any existing IP stack can be used as is: the protocol only interacts with routing table management.

In Figure 6.2, node N2, selected a few neighbor nodes in the network. These nodes will send node N2 packets. These selected nodes, N1 and N6 are called Multipoint Relays of node N2. Node N2 selects its MPR to cover all the nodes that are exactly two hops away from it. In our example: N7, N8, N9 and N4. A node which is not a Multipoint Relay can read the packet sent from N2 but cannot forward it.

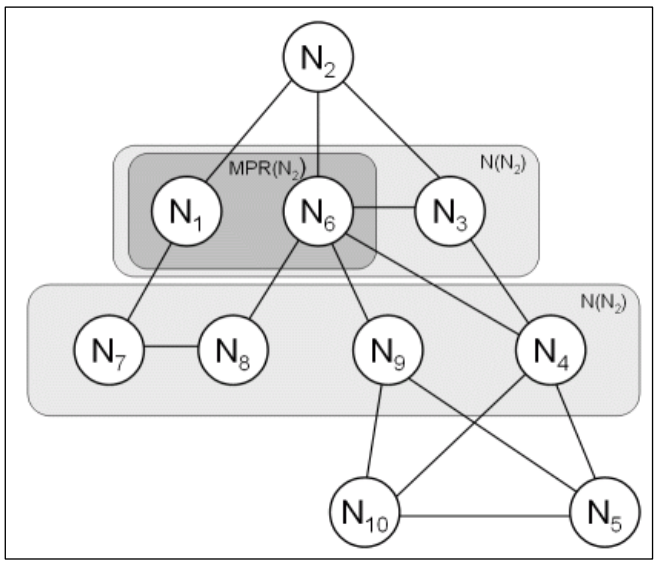


Figure 6.2: OLSR route selection.

Annex III –Kernel Programming

Linux kernel aspects

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

It is of course not possible go give an elaborate introduction to kernel programming in such a thesis. This introduction is supposed to teach the reader the very basics of Linux kernel programming. It shows the major aspects related to the implementation of the virtual network interface, namely to get a module built and have it registered with the central facilities of the Linux kernel. A more in-depth look at Linux kernel programming is given in [39].

Any structure or function which is referenced in the following can be looked up at the Cross Referencing Linux [40] project. They provide a search engine and a hyper-linked view of the Linux source code.

The Linux kernel

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

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

The Hello World Module

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

Listing 1 shows the implementation of this simple module. Even in this simplistic example a peculiarity of the Linux kernel shows up: the heavy usage of preprocessor macros. The first is

found after the “includes”. MODULE_LICENSE declares the license under which a module is distributed. As the Linux kernel itself is distributed only under GPL [41] which does not allow linking proprietary objects to GPL-objects, it is quite unavoidable to choose GPL as the module’s license. Otherwise many kernel features are hidden from the module, which is very restricting.

The next macro is encountered in line 8. KERN_ALERT, KERN_DEBUG, KERN_INFO and others define the class of a log message which is to be printed to the kernel log ring-buffer. These macros expand to strings which prefix the actual message.

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

Listing 2 shows the corresponding makefile and listing 3 how the module is loaded into the kernel. Looking at the output of the dmesg program should reveal the two strings.

```
01      #include <linux/module.h>   /* Needed by all modules */
02      #include <linux/kernel.h>    /* Needed by all modules */
03      #include <linux/init.h>
04      MODULE_LICENSE("GPL");
05
06      static int hello_init(void)
07      {
08      printk(KERN_ALERT "Hello, world\n");
09      return 0;
10      }
11
12      static int hello_exit(void)
13      {
14      printk(KERN_ALERT "Bye, world\n");
15      }
16
17      module_init(hello_init);
18      module_exit(hello_exit);
```

Listing 1: A minimal kernel module.

```
01      ifneq (($KERNELRELEASE),)
02      obj-m := hello.o
03      else
04      KERNELDIR ?= /lib/modules/$(shell uname -r)/build
05      PWD := $(shell pwd)
06
07      default:
08      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
09      endif
```

Listing 2: The Makefile

```
01      make
02      insmod hello.ko
03      rmmod hello
```

Listing 3: Building and loading/unloading the module.

The build system

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

Becoming part of the kernel

To let the module appear in the network section of the kernel configuration, its source has to be moved to *net/hello/* in the kernel source tree. The file *net/Kconfig* has to contain a line source *net/hello/Kconfig*. In the hello directory, a new file Kconfig has to be created according to listing 4. The *makefile* has to be adapted to its new environment as the module has to be compiled if and only if it is enabled in the configuration.

```
01      config HELLO
02                      tristate "Hello world module"
03                      ---help---
04                        To compile this code as a module,
05                        choose M here: the module
06                        will be called hello.
07
08                        If unsure, say N.
```

Listing 4: *Config* file for the kernel build system.

Listing 5 shows how it might look like. Apart from some general conventions for in-kernel *makefiles*, the main difference to the simple one in the previous section is in line 15 where the content of the variable CONFIG_HELLO is evaluated. This variable is set by the configuration system of the kernel and refers to the config HELLO directive in listing 4.

```
01      DEBUG = y
02
03      ifeq ($(DEBUG),y)
04      DEBFLAGS = -O -g -DHELLO_DEBUG
05      else
06      DEBFLAGS = -O2
07      endif
08
09      CFLAGS += $(DEBFLAGS) -I$(LDDINC)
10
11      TARGET = hello
12
13      ifneq ($(KERNELRELEASE),)
```

```
14
15      obj-$(CONFIG_HELLO)        := hello.o
16
17      #hello-objs :=  #no other objects are linked to hello.o
18
19      else
20
21      KERNELDIR ?= /lib/modules/$(shell uname -r)/build
22      PWD       := $(shell pwd)
23
24      modules:
25      $(MAKE) -C $(KERNELDIR) M=$(PWD) LDDINC=$(PWD) modules
26
27      endif
28
29
30      install:
31      install -d $(INSTALLDIR)
32      install -c $(TARGET).o $(INSTALLDIR)
33
34      clean:
35      rm -rf *.o *~ core .depend *.ko
36      rm -rf *.mod.c .tmp_versions .*.cmd
37
38
39      depend .depend dep:
40      $(CC) $(CFLAGS) -M *.c > .depend
41
42      ifeq (.depend,$(wildcard .depend))
43      include .depend
44      endif
```

Listing 5: Makefile using the kernel build system.

The Linux Device Model

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

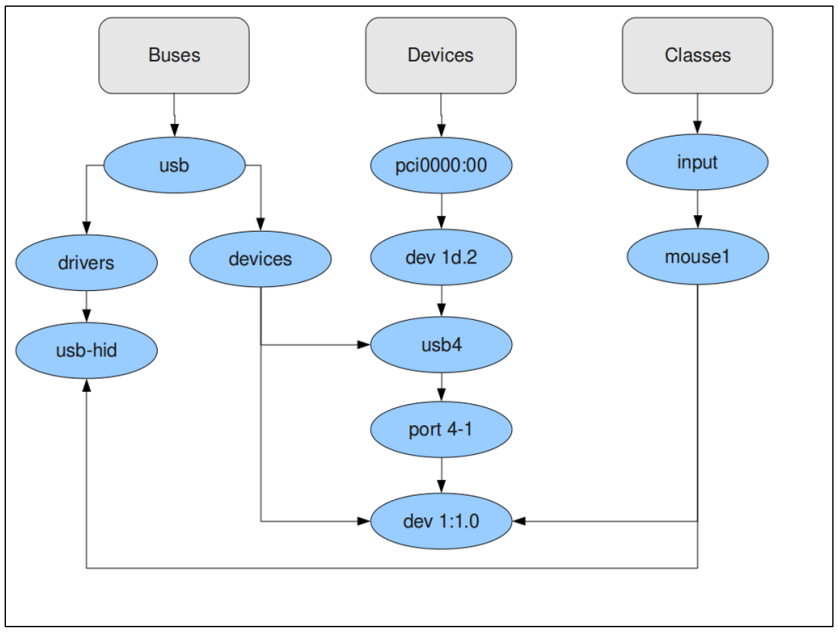


Figure 6.3: A look into the device model.

The kernel programmer's work became easier because all subsystems work similarly while at the same time the registration of a driver and its devices to the system may have become more difficult. Certainly, the new device model is a giant step in the development of the Linux kernel and shows that Linux 2.6 is a modern and well designed operating system. The device model is mainly split into buses, classes and devices. A small piece of it is shown in figure 6.3 which is adapted from [39].

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

Each object in the device model (e.g. device, driver, bus) is represented by a *kobject*. The *kobject's* tasks include reference counting, SysFS representation, hotplug event handling. It holds the device module structure together by having pointers to the parent, a *kset*, a list containing its children. It helps distinguishing the different types of *kobjects* with a pointer to a *kobj_type*. The *kobject* structure and its primary helpers *kobj_type* and *kset* are defined in "include/linux/kobject.h". An excerpt of this file is given in listing 6.

Listing 7 shows how a *kobject* is normally used. It is a member of the struct which wants to use *kobject's* facilities. This technique is encountered throughout the kernel in many places, e.g. the linked list implementation (cf. listing 6). At this point it might be important to know that the basic structure device of which any device in the kernel has an instance normally is not used

solely, instead each subsystem defines a container for this structure. Another interesting point is to see that the structure class_device contains a *kobject* and the structure device contains another one. The usefulness of this will be seen in the section covering SystemFS.

```
01      struct kobj_type {
02                      void (*release)(struct kobject *);
03                      struct sysfs_ops        * sysfs_ops;
04                      struct attribute        ** default_attrs;
05      };
06      struct kset {
07                      struct subsystem         * subsys;
08                      struct kobj_type          * ktype;
09                      struct list_head           list;
10                      struct kobject             kobj;
11                      struct kset_hotplug_ops  * hotplug_ops;
12      };
13      struct kobject {
14                      char                    * k_name;
15                      char                    name[KOBJ_NAME_LEN];
16                      struct kref               kref;
17                      struct list_head          entry;
18                      struct kobject          * parent;
19                      struct kset             * kset;
20                      struct kobj_type       * ktype;
21                      struct dentry           * dentry;
22      };
```

Listing 6: kobject structures.

```
01      struct class_device {
02                      struct list_head        node;
03                      struct kobject          kobj;
04                      struct class            * class;
05                      struct device           * dev;
06                      void                    * class_data;
07                      char    class_id[BUS_ID_SIZE];
08      };
```

Listing 7: A kobject consumer.

Registering with the device model

Devices in Linux 2.6 normally will not be created out of the blue. The need for a device structure or even the whole driver arises when a device is detected through *hotplug* events or probing on the bus. Only special devices like the pure virtual network interface have to be initialized and registered with their respective subsystems manually. Another exception are busses which define on the one hand a *bus_type* and on the other hand a device.

Busses like the platform bus which do not have a physical representation (e.g. the USB has a representation in form of a UHCI controller) have to initialize their device structures themselves.

Network devices register with the network system through *register_netdevice* which also

includes a registration with the class net. It is important to distinguish between the different institutions to which a device might register and to know the various structures needed to do this. Table 6.1 tries to give an overview of the major entry points to the device model. It shows the relationship of subsystems and their structures and registration functions.

Part of The Model	***Structure***	***Function***
class	struct class_device	class_device_register
network subsystem	struct net_device	register_netdevice
bus	struct device	device_register

Table 6.1: Registration facilities of the device model.

System FS

The user interface to the new device model is a filesystem which is usually mounted in */sys*. It lists all devices, drivers, busses and their relations. All of them can export attributes which then are listed as files. Links to other parts of the system are implemented as directories, i.e. the pci bus contains directories representing the actual busses (pci controller) which again contain links to the connected devices.

Device directories contain links to their drivers. Via this filesystem the user or system utilities can access and modify the parameters of devices and drivers. The user can manipulate it using *echo*, *cat* and similar tools. The library *libvi* and the management tool *victl* make use of the files in the sys filesystem. Looking at Listing 8 one might be able to detect the correlation of the SysFS tree and the data structure showed in figure 6.3. As attributes are exported to *SysFS* as files, one has to define functions for read and write operations. The kernel provides macros and a convenient API to decorate a kobject[1] with custom attributes. A directory containing several attributes is attached to an existing kobject as shown in listing 9. The function *add_myattrs* is usually called upon initialization of the *class_device* structure. The functions providing the read and write operations on the *SysFS* files should return the number of bytes read or written. The network devices, which are covered later, contain a *class_device* structure. Considering the device eth0 the new attributes would appear in */sys/class/net/eth0/myattrs/*. The *class_device* also holds a link to a device structure which essentially is the the basic representation of any

[1] A kobject is an object of type struct kobject. Kobjects have a name and a reference count. A kobject also has a parent pointer (allowing kobjects to be arranged into hierarchies), a specific type, and, perhaps, a representation in the sysfs virtual filesystem.

device. This structure maintains all the connections to busses, drivers and some very specific information on power management, DMA and other things we usually do not want to get in touch with. This example shows how a *class_device* can be extended by attributes. This is what we usually want; as there lies the information a user/administrator has to deal with. The attributes exported by device cover mostly the mentioned low-level details which are usually read-only.

```
/sys
|-- block
       |-- ...
|-- bus
|   |-- ...
|   |-- pci
|   |   |-- devices
|   |   |   |-- ...
|   |   |   |-- 0000:00:1d.1 -> ...
|   |   |   |-- ...
|   |   `-- drivers
|   |       |-- Intel ICH
|   |       |   |-- 0000:00:1f.5 -> ...
|   |       |   |-- ...
|   |       |-- ...
|   |       `-- uhci_hcd
|   |           |-- 0000:00:1d.0 -> ...
|   |           |-- 0000:00:1d.1 -> ...
|   |           |-- ...
|   |-- ...
|   `-- usb
|       |-- devices
|       `-- drivers
|-- class
|   |-- input
|   |   |-- input0
|   |   |   |-- device -> ...
|   |   |   |-- ...
|-- devices
|-- firmware
|-- kernel
|-- module
`-- power
```

Listing 8: Look into SysFS.

```
static ssize_t store_attr_a(struct class_device *cd,
                                                    const char *buf,
                                                    size_t len)
{...}

static ssize_t store_attr_b(...){}
static ssize_t store_attr_c(...){}

static ssize_t show_attr_c( struct class_device *cd,
                                                    const char *buf,
                                                    size_t len)
{...}

static CLASS_DEVICE_ATTR(attr_a, S_IWUSR,
                                                    NULL, store_attr_a);
static CLASS_DEVICE_ATTR(add, S_IWUSR,
                                                    NULL, store_attr_b);
static CLASS_DEVICE_ATTR(maxdiff, S_IWUSR | S_IRUGO,
                                                    show_attr_c, store_attr_c);
```

```
20
21      static struct attribute *myattrs[] = {
22        &class_device_attr_attr_a.attr,
23        &class_device_attr_attr_b.attr,
24        &class_device_attr_attr_c.attr,
25        NULL
26      };
27
28      static struct attribute_group mygroup = {
29        .name = "myattrs",
30        .attrs = myattrs,
31      };
32
33      int add_myattrs(struct class_device *dev)
34      {
35              struct kobject *kobj = &dev.kobj;
36              int err;
37              err = sysfs_create_group(kobj, &mygroup);
38              if(err)
39              {
40                      pr_info(%s can't create group %s\n",
41                              __FUNCTION__, mygroup.name);
42
43              }
44              return err;
45      }
```

Listing 9: Adding a group of attributes.

The Network subsystem

The Linux network subsystem breaks with the UNIX philosophy of everything being a file. Contrary to block and char devices, network devices do not have an entry point in the */dev* directory. There is usually no reason to perform read or write operations on a network device. These operations are performed on a socket, of which many hundreds can be multiplexed to a network interface. A network interface has to provide means for transmitting and receiving packets. The network subsystem is completely independent of protocols (either hardware or software) albeit providing major support for ethernet devices and the TCP/IP protocol suite.

Implementing a device similar to an ethernet device is very tempting, so that even the plip[1] device, which is a network device that links two computers via their parallel ports, resembles an ethernet device in many ways.

Initialization

A network device is created using the function *alloc_netdev* and registered with the network subsystem using *register_netdev*. The usage of these functions is demonstrated in Listing 11. The function *mydev_create* carries out all steps necessary to create a new network device. The

[1] The Parallel Line Internet Protocol (PLIP) is an encapsulation of the Internet Protocol designed to work over a personal computer parallel port via a null-printer cable, sometimes called a “laplink” cable.

structure type *mydev_private* is the place where device specific data is stored. The function *alloc_netdev* allocates the space for this private data, too. It is in fact appended to the structure *net_device*. The pointer *priv* links to the start of the private data. Listing 11 also shows how the *class_device* structure discussed earlier is used in a specific subsystem.

```
01      struct mydev_private{
02              /* private fields */
03      };
04
05      void mydev_setup(struct net_device *dev)
06      {
07              /* custom initialization code */
08      }
09
10      struct net_device *mydev_create()
11      {
12              mydev = alloc_netdev(sizeof(struct mydev_private),
13                                   "mydev%d", mydev_setup);
14              if(mydev)
15              {
16                      if(register_detdev(mydev))
17                      {
18                              /* error handling */
19                              free_netdev(mydev);
20                      }
21              }
22              else
23              {
24                      /* error handling */
25              }
26      }
```

Listing 10: Initialization of a network device.

```
01      struct net_device *alloc_netdev(
02              int sizeof_priv,
03              const char *name,
04              void (*setup)(struct net_device *));
05      void free_netdev(struct net_device *dev);
06      int register_netdev(struct net_device *dev);
07      void unregister_netdev(struct net_device *dev);
08
09      struct net_device
10      {
11              ...
12              void *priv;
13              ...
14              struct class_device class_dev;
15              ...
16      }
```

Listing 11: Main network device infrastructure.

Default interface

The network subsystem requires a device to implement a set of default functions. During initialization, the driver has to store the pointers to the implementing functions into the appropriate fields of the *net_device* structure (cf. Listing 10). Not all of these functions have to be implemented specifically, as the kernel includes some default implementations which are enabled through *netdev_alloc*. A list of the most common candidates for custom implementation is given in Listing 12.

```
01      int   (*open)(struct net_device *dev);
02      int   (*stop)(struct net_device *dev);
03      int   (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
04      int   (*set_mac_address)(struct net_device *dev, void *addr);
05      int   (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
06      int   (*set_config)(struct net_device *dev, struct ifmap *map);
07      int   (*change_mtu)(struct net_device *dev, int new_mtu);
08      void  (*tx_timeout) (struct net_device *dev);
```

Listing 12: Network device interface service routines.

- **open**

  This function is called as soon as ifconfig activates the device. Any resources should be initialized at this point, i.e. in a physical device this includes IRQ, DMA, etc.

- **stop**

  This is the opposite to open.

- **hard_start_xmit**

  The actual workhorse of a network device which transmits packets. Will be covered in-depth in the next section.

- **set_mac_address**

  If a device is able to set its mac address, e.g. in a register of the chip on the adapter, then this function would perform this low-level work. The default implementation just sets the corresponding field net_device->dev_addr.

- **do_ioctl**

  Only if the interface is desired to perform specific ioctl operations this field must be non-null. The implementation of custom ioctl operations is not covered in this thesis[1].

- **change_mtu**

  If the MTU for this interface changes, this function is called.

- **tx_timeout**

  If a packet transmission fails to be completed within reasonable time, this function is supposed to handle the problem and to resume transmission.

[1] With the introduction of sysfs the ioctl mechanism has become obsolete in most cases, since every new ioctl operation is like a new system call. The kernel API therefore changes rapidly and becomes complex. Further, ioctl operations are assigned global numbers which have to be coordinated to not overlap between different devices.

Packet transport

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

The protocol-independency of the *sk_buff* structure in Listing 14 is clearly visible in the excessive usage of unions. This structure perfectly fits the packet-oriented nature of most modern network protocols. It integrates the header-data for three protocol-layers, the actual payload and a lot of administrative information. The latter mostly relate to the packet filter (*Netfilter*) and caching. The *sk_buff* system includes several functions to manipulate this non-trivial structure. Listing 13 shows the signature of these functions.

```
01      struct sk_buff *skb_clone(struct sk_buff *skb, int priority);
02      struct sk_buff *alloc_skb(unsigned int size, int priority);
03      void skb_trim             (struct sk_buff *skb, unsigned int len)
04      unsigned char *skb_pull   (struct sk_buff *skb, unsigned int len);
05      unsigned char *skb_push   (struct sk_buff *skb, unsigned int len);
06      unsigned char *skb_put    (struct sk_buff *skb, unsigned int len);
```

Listing 13: sk_buff manipulation.

- **skb_clone**

  Duplicates an sk_buff structure in its entirety

- **alloc_skb**

  Allocates a sk_buff structure. This is mainly used in the receiving part of a network driver.

- **skb_pull**

  Removes data from the start of the buffer. This function returns a pointer to the next data in the buffer. Subsequent calls to skb_push will overwrite the old data.

- **skb_push**

  Adds data to the start of the buffer. A pointer to the new start is returned.

- **skb_put**

  Adds data to the end of the buffer. A pointer to the start of the extra data is returned.

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

Listing 14: Excerpt of the sk_buff structure.

Annex IV – Installation Tutorial

Installation Tutorial

In this annex it is explained how to install and run the implemented module. For that is necessary to meet the following series of requisites:

- Computer running Debian GNU/Linux operating system;
- Kernel version 2.6, or higher;
- The latest version of Sysfs installed;
- The package Bridge-utils installed.

Linux kernel

1. Download the source of the kernel version (must be at least version 2.6) from www.kernel.org;
2. Extract the folder to the */usr/src* directory;
3. Copy the config-linux.your.linux.version from */boot* directory to */usr/src/linuxversion*
4. Change the name of the config-linux.your.linux.version to ".config";
5. Go to */usr/src/linuxversion* and execute the following command: "make menuconfig" then choose the option load an alternate version, press ok and then exit;
6. Copy the file socket.c to the */usr/src/linuxversion/net* directory;
7. Copy the file *dev.c* to */usr/src/linuxversion/net/core* (this step is not necessary with the netfilter hooks version);
8. Copy the files netdevice.h and sockios.h to the */usr/src/linuxversion/include/linux* directory;
9. Go to the */usr/src* directory and execute the following commands:
    a. make bzImage
    b. make modules
    c. make install
    d. make modules_install
    e. mkinitramfs –o /boot/initrd.img-$(uname -r) 'kernel version' (Ex: mkinitramfs –o /boot/initrd.img-$(uname -r) 2.6.31.14)
10. Add a line with the new kernel version to the */boot/brug/grub.cfg* file;

```
menuentry "Ubuntu, Linux 2.6.31-14 for VI" {
                recordfail=1
                if [ -n ${have_grubenv} ]; then save_env recordfail; fi
        set quiet=1
        insmod ext2
        set root=(hd0,4)
        search --no-floppy --fs-uuid --set 5d7a1424-cf4e-4c2c-888f-7199a7e908c0
        linux   /boot/vmlinuz-2.6.31.14 root=UUID=5d7a1424-cf4e-4c2c-888f-7199a7e908c0 ro
        quiet splash
        initrd /boot/initrd.img-2.6.31-14
        }
```

11. Reboot the computer and choose the new kernel version;

Routing protocols

1. Get the source distribution of AODV-UU [43];
2. Install according to the AODV-UU installation manual;
3. Get the source distribution of OLSRD [9];
4. Install according to the OLSRD installation manual.

Libsysfs

Install the development package of libsysfs for your distribution or get and install the source distribution from [42].

Virtual interface

1. Unzip kmod.zip and in that directory execute the following commands:
   i. Make;
   ii. insmod vi.ko (to load the module).
2. Go the directory of the victl executable and run the following command: chmod a+x victl (to give the needed permissions);
3. To run the virtual interface execute the command: ./victl;
4. Do debug eventual errors use the command: dmesg.

Usage

The *victl* is self-explanatory. The following is an example on how to add a virtual interface and associate some existing network interfaces:

1. Add a virtual interface using *victl addvi vi0;*
2. Start the interface by *ifconfig vi0 up;*
3. Add an existing network interface to the virtual interface by *victl addif vi0 eth1;*
4. Set the priority by *victl setportprio vi0 eth1 "priority"*;
5. Add another existing interface by *victl addif vi0 wlan0;*
6. Set a MAC address for the virtual interface by *ifconfig vi0 hw ether $MAC;*
7. Set an IP address by *ifconfig vi0 $IP or dhclient vi0*.

Annex V – Testbed Configuration

File: **/etc/config/network**

```
config 'interface' 'loopback'
option 'ifname' 'lo'
option 'proto' 'static'
option 'ipaddr' '127.0.0.1'
option 'netmask' '255.0.0.0'
config 'interface' 'lan'
option 'type' 'bridge'
option 'proto' 'static'
option 'ipaddr' '192.168.2.1' #Each router has a different ip
option 'netmask' '255.255.255.0'
option 'ifname' 'eth0.0'
config 'interface' 'wan'
option 'ifname' 'eth0.1'
option 'proto' 'dhcp'
config 'interface' 'adhoc'
option 'ifname' 'ath0'
option 'proto' 'static'
option 'ipaddr' '192.168.0.1' #Each router has a different ip
option 'netmask' '255.255.255.0'
```

File: **/etc/config/system**

```
config system
option hostname APXX #XX is the number of each AP (e.g. 01, 02, 03)
option timezone UTC
config button
option button reset
option action released
option handler "logger reboot"
option min 0
option max 4
config button
option button reset
option action released
option handler "logger factory default"
option min 5
option max 30
```

File: **/etc/config/wireless**

```
onfig 'wifi-device' 'wifi0'
option 'type' 'atheros'
option 'disabled' '0'
option 'mode' '11a'
option 'agmode' '11bg'
option 'maxassoc' ''
option 'distance' ''
option 'diversity' '1'
option 'txantenna' '0'
```

```
option 'rxantenna' '0'
option 'antenna' ''
option 'channel' '07'
config 'wifi-iface'
option 'device' 'wifi0'
option 'network' 'adhoc'
option 'mode' 'ahdemo'
option 'encryption' 'none'
option 'bssid' '02:00:01:02:03:04'
option 'server' ''
option 'port' ''
option 'hidden' '0'
option 'isolate' '0'
option 'txpower' '18'
option 'bgscan' '0'
option 'frag' ''
option 'rts' ''
option 'wds' '0'
option 'key1' ''
option 'key2' ''
option 'key3' ''
option 'key4' ''
option '80211h' ''
option 'compression' ''
option 'bursting' ''
option 'ff' ''
option 'wmm' ''
option 'xr' ''
option 'ar' ''
option 'turbo' ''
option 'macpolicy' 'none'
option 'ssid' 'VirtualInterfaceTest'
```

Annex VI – Energy Consumption Script

```
#!/bin/sh
#power.sh

COUNTER=0

while [ $COUNTER -lt 60 ]; do

awk 'NR==5{print >> /home/mota/Desktop/teste.txt}'
/proc/acpi/battery/BAT0/state
let COUNTER=COUNTER+1
echo $(date)
sleep 10

done
```