Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2012

André Filipe Ferreira Prata

Gestão de ligações baseada em IEEE 802.21

Connection management based on IEEE 802.21

Universidade de Aveiro
2012

Departamento de
Eletrónica, Telecomunicações e Informática

André Filipe Ferreira Prata

Gestão de ligações baseada em IEEE 802.21

Connection management based on IEEE 802.21

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Alexandre de Sousa Gonçalves, Professor adjunto da Escola Superior de Tecnologia e Gestão de Águeda da Universidade de Aveiro.

o júri / the jury

presidente / president	Professor Doutor António Luís Jesus Teixeira Professor associado da Universidade de Aveiro
vogais / examiners committee	Doutor Pedro Miguel Naia Neves Investigador da Portugal Telecom Inovação
	Professor Doutor Diogo Nuno Pereira Gomes Professor auxiliar convidado da Universidade de Aveiro
	Professor Doutor Pedro Alexandre de Sousa Gonçalves Professor adjunto da Universidade de Aveiro

agradecimentos / **acknowledgements**

O sucesso deste trabalho atribui-se também aos meus pais, aos professores Diogo Gomes e Pedro Gonçalves, ao Daniel Corujo e ao Carlos Guimarães.

Palavras Chave

linux, 802.21, gestão de ligações

Resumo

Avanços recentes nas telecomunicações conduziram a uma combinação de várias interfaces de acesso à rede num único dispositivo. Os programas de gestão de ligações atuais lidam com as diferentes tecnologias individualmente, e baseiam a seleção da rede de acesso em parâmetros tais como a potência de sinal, ou taxa de transmissão máxima. Nem sempre estes mecanismos refletem a performance real de uma rede, levando a uma experiência de acesso fraca.

Neste trabalho é implementada uma *framework* de gestão de ligações inovadora, baseada na norma IEEE 802.21. Esta norma disponibiliza mecanismos que facilitam e otimizam *handovers* entre diferentes tecnologias e a seleção de ligações através da troca de informações entre as entidades da rede e o terminal, incluindo informação de QoS, desempenho ou outras características. Além disso, a norma permite a gestão de dispositivos independentemente da tecnologia, através de uma interface uniformizada ao nível da camada de ligação de dados.

Em virtude da extensão desta interface com mecanismos multi-camada, a nova *framework* possibilita a configuração asbtrata das interfaces de rede, incluindo a associação, configurações de segurança e endereçamento IP. O acesso a informação da rede capacita ainda os gestores de ligações para a realização de melhores decisões, tendo em conta o estado da rede e os requisitos das aplicações do terminal.

Esta *framework* é integrada com as ferramentas e applets de configuração de rede do sistema operativo GNU/Linux, através da substituição transparente da aplicação *NetworkManager*. Em comparação, a nova *framework* apresenta *overhead* insignificante, uma quantidade de código inferior e melhor consumo de bateria, além de mecanismos otimizados para ligação oportunística.

Keywords linux, 802.21, connection management

Abstract

Recent advances in telecommunications have lead to the combination of various network access interfaces in a single device. Current network management software handles different technologies individually, and base connection decisions on parameters such as signal strength, or maximum throughput. Often, these network attributes do not reflect the real network perfomance, leading to poor network experience.

In this work, a novel network management framework is implemented, based on the IEEE 802.21 standard. This standard provides mechanisms to facilitate and optimize inter-technology handovers and network selection through information exchanges between network and terminal entities, including QoS and other network capability and performance information. Moreover, it enables media independent device management via a common link layer interface.

By extending this interface with cross-layer mechanisms, the new framework allows abstract configuration of the network interfaces, including network association, security setup procedures and IP address configuration. The access to network information will additionally empower network managers to perform better decisions that take network state and terminal application requirements into account.

This framework is integrated with the existing configuration tools and applets from the GNU/Linux Operating System, by seamlessly replacing the existing *NetworkManager* application. In doing so, the new framework shows insignificant overhead, a reduced code base and better battery consumption, on top of optimized procedures for opportunistic network attachment.

Contents

List of Figures

List of Tables

List of Acronyms

3GPP	3rd Generation Partnership Project
AAA	Authentication, Authorization and Accounting
ABC	Always Best Connected
AES	Advanced Encryption Standard
AP	Access Point
API	Application Programming Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
BS	Base Station
BSS	Basic Service Set
CCMP	Counter Cipher Mode with Block Chaining Message Authentication Code Protocol
CID	Connection Identifier
CORBA	Common Object Request Broker Architecture
CoS	Class of Service
CPU	Central Processing Unit
CQM	Connection Quality Monitor
CRDA	Central Regulatory Domain Agent
DCOP	Desktop Communication Protocol
DE	Desktop Environment
DHCP	Dynamic Host Configuration Protocol
DHCPv4	DHCP version 4

DHCPv6	DHCP version 6
DNS	Domain Name System
DSL	Digital Subscriber Line
EAP	Extensible Authentication Protocol
EAPoL	EAP over LAN
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESS	Extended Service Set
ETSI	European Telecommunications Standards Institute
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GPRS	General Packet Radio Service
GSM	Global System for Mobile communication
GTK+	GIMP Toolkit
GUI	Graphical User Interface
HESSID	Homogeneous ESS Identifier
IBSS	Independent Basic Service Set
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IE	Information Element
IPC	Inter-Process Communication
IP	Internet Protocol
IPsec	Internet Protocol Security
IPv4	IP version 4
IPv6	IP version 6
ISC	Internet Systems Consortium
KDE	K Desktop Environment
LAN	Local Area Network

LEAP	Lightweight Extensible Authentication Protocol
LLC	Logical Link Control
LSAP	Link Service Access Point
LTE	Long Term Evolution
LXDE	Lightweight X11 Desktop Environment
MAC	Media Access Control
MIB	Management Information Base
MICS	Media Independent Command Service
MIHF	Media Independent Handover Function
MIH	Media Independent Handover
MIIS	Media Independent Information Service
MN	Mobile Node
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NTP	Network Time Protocol
OS	Operating System
PDU	Protocol Data Unit
PID	Process ID
PIN	Personal Identification Number
PKI	Public Key Infrastructure
PMK	Pairwise Master Key
PMKSA	Pairwise Master Key Security Association
PoA	Point of Attachment
PoS	Point of Service
PUK	Personal Unblocking Code
QoS	Quality of Service
RADIUS	Remote Authentication Dial In User Service

RAII	Resource Acquisition Is Initialization
RC4	Rivest Cipher 4
RSSI	Received Signal Strength Indicator
SAP	Service Access Point
SIM	Subscriber Identity Module
SNR	Signal to Noise Ratio
SINR	Signal over Interference plus Noise Ratio
SMS	Short Message Service
SNMP	Simple Network Management Protocol
SSID	Service Set Identifier
SS	Subscriber Station
TCP	Transmission Control Protocol
TKIP	Temporal Key Integrity Protocol
TLV	Type-Length-Value
TOS	Type of Service
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
URL	Uniform Resource Locator
USB	Universal Serial Bus
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
WEP	Wired Equivalent Privacy
WEs	Wireless Extensions
WIMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network
WMAN	Wireless Metropolitan Area Network
WPA2	Wi-Fi Protected Access 2

WPA	Wi-Fi Protected Access
WPS	Wi-Fi Protected Setup
XML	Extensible Markup Language

Chapter 1

Introduction

The Internet has steadily gained popularity from its first days to the point that, in 2011, one third of the world population was estimated to be using it [1]. This fact is attributed to the widespread use of computers and, more recently, mobile phones. These devices enable virtually instant access to information, entertainment as well as voice and video communications across the globe.

Various physical and logical data communication technologies emerged for interconnecting these devices. Nowadays, most computers and smartphones offer at least two network access interfaces. Computers can also be coupled with after-market dongles for access to additional network technologies. Some technologies are focused on high throughput for stationary devices, while others attempt to provide mobile access in geographically broad areas. Various technologies combined in a single device complement each other, enabling the device to adapt to the network scenarios that better support its purpose.

1.1 Motivation and Goals

The combination of multiple network technologies in a single device requires a management entity for controlling each device and chose, at any given time, the best interface for network access. To achieve this, modern Operating Systems (OSs) usually have a component called a Network Manager. Network Managers offer a high level interface to control the physical and logical properties of the network interfaces, and usually provide Graphical User Interfaces (GUIs) for users to configure network parameters and connectivity preferences.

Currently, there is a myriad of connection management software for mobile terminals. They are usually distributed as pre-installed software by Operating Systems or device vendors, and also integrated with device driver bundles, provided by network interface vendors and operators. When provided by Operating Systems, these programs are meant as generic solutions that attempt best at keeping an active connection with every available interface (although it is usually possible to configure mutual exclusion). Operators, on the other hand, develop custom software in order to introduce added value associated with their provided services. Depending on the operator, this includes the ability to track data usage, place calls or Short Message Service (SMS) interactions, locate operator hotspots, etc. The following set of issues can be detected in virtually every connection management solution:

- **OS portability**: there is a multitude of available Operating Systems that provide different programmable interfaces and technologies for network management. While there are software technologies that allow high level programs to run in different OSs, network management software must deal with low level aspects of the machine for which there are no OS-independent frameworks.
- **Technology independence**: network manager implementations are tied to specific hardware technologies as much as they are to the OS on which they are deployed. Different technologies require different control primitives, which in turn result in the need for different software drivers and programmable interfaces. As such, Network Managers are usually tailored for a few specific network technologies, leaving the others out of the supported feature list.
- **Network communication**: existing Network Managers usually take into account various metrics about each network in order to select the best possible connection, such as "signal quality", or maximum "bit rate". More often than not, these metrics result in bad network selection decisions, since the experience provided by a network greatly depends on a number of factors that clients cannot perceive, such as the load of a network endpoint. However, the network knows about these factors, and may issue information that helps clients better choose points of attachment. Some operators implement proprietary solutions to address this problem.
- **Application requirements**: there is no single "best network" for any given scenario. Depending on the applications and services a terminal is running, the perceived user experience may depend on available throughput, minimum latency, or other network policies. These variables seldom determine network selection, as this information is not available prior to attachment to a target network.

With different kinds of access technologies available to multi-interface mobile terminals, achieving an optimal network selection decision depends on multiple parameters [2], ranging from:

1. The dynamics of the wireless strata (e.g., Signal to Noise Ratio (SNR), available bandwidth, cell load);
2. Requirements placed by the service content being accessed (e.g., minimum latency);
3. Requirements placed by the user (e.g., perceived video and/or audio quality, cost);
4. Network conditions (e.g., cell load, requested service, policies).

The different criteria involved must not only take into consideration the capabilities of the service being provided, but also the resources available in the network and, ultimately, the user satisfaction. As such, optimal decisions have the need to assess different objectives, from different layers of the network stack, in order to achieve an Always Best Connected (ABC) [3] solution. In this sense, different schema are possible, varying between mobile terminal centric decisions [4], network controlled decisions [5], or even combinations of both where the perspective of the terminal and network come together to optimize the handover decision to a broader set of requirements [6].

A new IEEE standard, by the 802.21 [7] group, defines a technology independent interface for handover optimization. This interface targets link layer operations, and standardizes communication between network and terminal entities in order to facilitate decisions regarding network handovers. The goal of this dissertation is to leverage from this standard, extending it when necessary, in order to provide an Enhanced Media Independent COnnection Manager (EMICOM) framework that will enable enhanced network management with the previously enumerated issues in mind. The framework is to be integrated with the GNU/Linux desktop OS by seamlessly replacing the most widely used Network Manager for the platform.

1.2 Document outline

The remainder of the document is structured as follows. Chapter 2 introduces some of the most common network technologies and requirements for Internet access. Chapter 3 exposes the GNU/Linux platform from the point of view of a programmer targeting the control of network devices and protocols, as well as the integration with other desktop applications. This is followed by an overview of the existing network management tools for the platform. Chapter 4 presents the new IEEE 802.21 standard and an open-source implementation of the protocol. Some required extensions to the protocol are proposed in the same Chapter, in order to allow network management tasks. The developed framework is described in Chapter 5, and evaluated in Chapter 6. Finally, Chapter 7 concludes and gives an insight of future work based on this dissertation.

Chapter 2

Accessing the Internet

In telecommunications, passing information from one point to another comprises a complex series of interactions. Any group of devices passing information between one another is said to form a network. Two networks can be interconnected to form a larger network; this is the origin of the term Internet (short for Internetwork); it is a large network of networks that is supported by a considerably large amount of devices (hardware) and programs (software). The Internet architecture is based on a layered stack that isolates protocols in different layers [8]; new protocols can be introduced in one layer without affecting other layers or protocols. When data is sent from one device to another on a network, it travels down these layers in the form of *packets*, which are then physically transported to the destination, where they will travel up the stack until they arrive at the intended program. At the top of the protocol stack is the client of the Internet itself, the software that is looking to transmit data to another host; the bottom concerns to the device that effectively transmits the data through the physical medium. In between are the following four layers considered by the Internet architecture [8]:

- **Application** (or Layer 5): software clients of the Internet design their own protocols for distributed communication, such as the File Transfer Protocol (FTP), for transferring files over the Internet.

- **Transport** (or Layer 4): several applications can use the Internet concurrently from the same system. The protocols in the transport layer assign each application a number, called *port*, that allows unequivocal delivery of packets to the intended client. Such is the case of the Transmission Control Protocol (TCP), whose goal is also to implement a reliable, ordered delivery of packets between clients.

- **Internet** (or Network Layer, or Layer 3): the Internet Protocol (IP) itself is defined in this layer. It solves the problem of correctly forwarding the packets, if necessary through intermediate routers, to the desired recipient. As the name points out, it is the core of the Internet, and will be further detailed in section 2.3.

- **Link** (or Layer 2): this layer provides means to address devices physically, through Media Access Control (MAC) addresses, as well as controlling the access to the shared, physical medium, including collision detection. MAC addresses are uniquely attributed by device vendors, but some devices allow to manually assign different addresses. This layer usually imposes a limit on frame size for transmission, labeled the Maximum Transmission Unit (MTU); the Network layer handles fragmentation for packets larger than this limit.

Many devices support Internet access, the most generally known to the user being the computers, smartphones and tablets. Acquiring network connectivity encompasses the configuration of certain aspects of the Internet Protocol. Some networks also employ security mechanisms that hosts must support. However, the first step towards network connectivity is the physical setup of the devices, which varies with technology.

2.1 Network Device Technologies

Network devices are mainly grouped in two categories: wired and wireless. Wired devices usually communicate with each other through a cabled medium, the most common being based on copper. Fiber optics is replacing the copper for the newer, higher performing networks, based on guided beams of light instead of electric current. Wireless devices rely on antennas that irradiate electromagnetic waves to broadcast information over the air.

The first requirement for network connectivity in wired technologies is simple: correctly setup the cables between the devices. Different standards define the requirements that the network cables must conform to, such as maximum length and impedance; as long as the cables conform to these requirements, attaching a cable will enable physical connectivity on a network.

Attaching wireless nodes introduces further complexity, because electromagnetic signals have a limited range that depends on the power of the transmitting device, the sensibility of the receiving interface, and the physical elements or obstacles in between. Electromagnetic signal strength decreases with range, and there is no way to evaluate the availability of signal range in the same way a cable attachment is determined. However, it is possible to measure the strength of a signal, as perceived by the receiving device. The usual metric for the "present signal" is the Watt. Since the strength of an electromagnetic signal is inversely proportional to the square of the distance from the source, the strength is best represented in a logarithmic scale of the measured Watt value. This is called $dBmW$ (or just dBm), for decibel of the measured power referenced to one milliWatt (mW), represented by the formula $dBm = 10\log(mW)$. The amount of signal energy that determines whether communication is possible varies with specific devices and network technologies.

Networks are also grouped in respect to their geographical span. It is common to consider Personal, Local, Metropolitan and Wide Area Networks. Wireless Metropolitan and Wide Area Networks, including IEEE[1] 802.16 and Cellular technologies such as 3rd Generation Partnership Project (3GPP)[2] standards, offer greater mobility due to their wider geographical

[1]Institute of Electrical and Electronics Engineers, `http://www.ieee.org/`

[2]3rd Generation Partnership Project, `http://www.3gpp.org/`

coverage. Local Area Networks (LANs) usually deliver higher bandwidths, and are very common at home and office environments, the most relevant being the IEEE 802.3 and 802.11 technologies.

2.1.1 IEEE 802.3

IEEE 802.3 [9] is the working group defining the physical and link layer's MAC of the wired network technology known as Ethernet. Many Ethernet standards were introduced over the years, improving on the network capacity past the gigabit per second throughputs. The first versions relied on coaxial cable buses, later replaced with twisted pairs and optical fiber cabling.

The technology may be used to create a direct connection between two devices, but the infrastructure targets more complex setups of many hosts interconnected by devices such as Hubs and Switches. Hubs are used to connect multiple network interfaces together, making them act as a single network segment. Switches commute packets between segments, depending on link layer packet destination address, thus preventing unnecessary bus occupation on the other segments.

2.1.2 IEEE 802.11

IEEE 802.11 [10] is the set of standards for implementing Wireless Local Area Network (WLAN) computer communication, from which the commercial Wi-Fi technology originates. Instead of cables, these networks operate over the electromagnetic spectrum, namely in the 2.4, 3.6 and $5GHz$ frequency bands, each divided in several channels.

A group of Stations communicating with one another over 802.11 forms a Basic Service Set (BSS). When all of the stations in the BSS are mobile stations and there is no connection to a wired network, the BSS is called Independent Basic Service Set (IBSS). An IBSS is a typically short-lived network, with a small number of stations, that is created for a particular purpose. In this mode (IBSS) there is no single master, and every station can directly communicate with each other. When a BSS includes an Access Point (AP), it is called Infrastructure BSS. When there is an AP, if one mobile station in the BSS must communicate with another mobile station, the communication is sent first to the AP and it is retransmitted from the AP to the destination mobile station.

Multiple interconnected BSSs form an Extended Service Set (ESS), where the APs communicate among themselves to forward traffic from one BSS to another, and to facilitate the movement of mobile stations between BSSs. ESSs are identified by a Service Set Identifier (SSID), which is a chosen string of octets with the maximum size of 32 bytes. Wi-Fi is connectionless and contention-based, meaning that there is no guaranteed logical link between a station and an AP, and that stations dispute the right to send packets every time they wish to.

Stations register with the AP in order to establish layer 2 connectivity. This process comprises two phases: authentication and association. If a station desires to attach to a different

AP in the same ESS, it roams by performing a reassociation to the new AP. Associating and Disassociating is the Ethernet equivalent of attaching and detaching a cable.

Stations know APs exist because they send periodic beacons. Sensing the electromagnetic spectrum for these beacons is called passive scanning. Active scans are performed by actively broadcasting probe request frames on the desired channels.

The 802.11 standard defines an arbitrary unit for signal measurement, called Received Signal Strength Indicator (RSSI). Vendors are free to define an RSSI_Maximum value, within the one byte range (up to 256 different levels), and there is no standard mapping between a dBm value and the resulting RSSI metric. Regardless of the vendor, it is common to derive a percentage metric of the signal strength by dividing current RSSI values by the arbitrary RSSI_Maximum (and then multiplying by 100). Although percentages do not translate to any dBm or mW value, a vendor would acknowledge that a perceived signal equal to RSSI_Maximum (100%) would be great, and assign to 0% the inability to sense any energy at all.

2.1.3 IEEE 802.16

The IEEE 802.16 group defines the standards for Wireless Metropolitan Area Networks (WMANs), resulting in a technology commercialized under the name Worldwide Interoperability for Microwave Access (WIMAX). The standard was developed to deliver connectivity between Subscriber Stations (SSs) and Base Stations (BSs) with typical non-line-of-sight cell radius of three to ten kilometers in the 2 to $11GHz$ frequency bands, and line-of-sight distribution service in the 10 to $66GHz$ bands [11].

IEEE 802.16 is connection-oriented; all services, including inherently connectionless services, are mapped to a connection. This provides a mechanism for requesting bandwidth, associating Quality of Service (QoS) and traffic parameters, and other actions associated with contractual terms of the service. Connections are referenced with Connection Identifiers (CIDs), and may require continuously granted or on-demand bandwidth. Upon association with a BS, an SS is assigned three management connections in each direction:

- The basic connection, which is used for the transfer of short, time-critical physical and link layer control messages.
- The primary management connection, used to transfer longer, more delay-tolerant messages such as those used for authentication and connection setup.
- The secondary management connection, used for the transfer of standards-based management messages such as Dynamic Host Configuration Protocol (DHCP) and Simple Network Management Protocol (SNMP) messages.

In addition to these management connections, Subscriber Stations are allocated transport connections for the contracted services. Transport connections are unidirectional and assigned in pairs, to facilitate different uplink and downlink QoS and traffic parameters.

2.1.4 Mobile Broadband

The 3GPP develops mobile Internet access technologies with capacity and throughput enhancements, through radio interface changes and incremental improvements to existing mobile telephony core network (both packet and circuit-switched). Several technologies have been standardized by the group, including the General Packet Radio Service (GPRS), the Universal Mobile Telecommunications System (UMTS) and, more recently, Long Term Evolution (LTE).

GPRS, originally standardized by the European Telecommunications Standards Institute (ETSI)[1], is a 2.5G mobile communications technology that enables mobile wireless service providers to offer their mobile subscribers packet-based data services over Global System for Mobile communication (GSM) networks [12]. Common applications of GPRS include Internet access, intranet/corporate access, instant messaging, and multimedia messaging. It is a best-effort service, implying variable throughput and latency that depend on the number of other users sharing the service concurrently.

UMTS is a 3G mobile communications technology that offers higher throughput, real-time services, and end-to-end QoS [13]. It offers both connection-oriented and connectionless services for point-to-point and point-to-multipoint communication. UMTS network services have different QoS classes for various types of traffic: conversational (voice and video telephony), streaming (video on demand), interactive (web browsing, gaming) and background (e-mail, SMS).

LTE is the 3GPP proposal for simplifying the internal architecture of the system and reducing network costs and latencies by transiting from the previous packet and circuit-switched combined network to a pure packet-switched, all-IP network [14, 15]. LTE has introduced a number of new technologies when compared to the previous cellular systems, achieving what is called a "flatter architecture"; data is no longer routed by traversing a hierarchy from the originating user through multiple layers of aggregation to a central core, and then re-routed back out in a multilayer dis-aggregation hierarchy to the targeted user.

In terms of protocols and infrastructure, mobile broadband networks are still considerably more complex than the previous technologies. They are commonly accessible through cellphones, smartphones and tablets. Computers usually depend on peripheral modems attached by Universal Serial Bus (USB).

2.2 Security Mechanisms

Many network technologies offer security mechanisms in order to limit network access to a specified group of people or devices, or to prevent data inside a network from being accessed from the outside. This issue is more common in wireless networks, since the data is inherently broadcasted over the air and not concealed in cables. One basic measure for preventing access from someone outside a network is to ignore all traffic from a specified link layer addresses.

[1]European Telecommunications Standards Institute, `http://www.etsi.org/`

This is efficient, but not very effective, because link layer addresses can be forged. It also does not imply any sort of protection of the data being transported.

Mobile broadband technologies usually require Subscriber Identity Module (SIM) cards on host devices for authentication purposes. A SIM card has both a memory and a small processing unit [16]. Part of the memory is accessible by the user, but another part, written by the operator, is protected and contains a secret key. SIM cards are only accessible after being provided a Personal Identification Numbers (PINs), and are blocked after a certain number of wrong PIN inputs. A Personal Unblocking Code (PUK) enables reactivating the SIM card, but failing this code a number of times will render the device irrevocable unusable.

Other common security mechanisms rely on secret material that is provided to the network layer by supplicant software. The original IEEE 802.11 standard proposed a security algorithm that aimed at providing data confidentiality compared to that of a traditional wired network, named Wired Equivalent Privacy (WEP) [17]. The algorithm was very efficient and easy to implement in both hardware and software, making it ideal for link layer security. It was based on a stream cipher generated by the WEP Algorithm (based on Rivest Cipher 4 (RC4)), with the initial inputs being a fixed-length key (previously shared and secretly stored by both Stations and APs) and an Initialization Vector (changed as frequently as every packet). WEP was found to contain several weaknesses [18, 19, 20], which were incorporated into software that helps to gain access to a network in a matter of seconds[1].

In its draft version, the IEEE 802.11i amendment proposed a new algorithm called Wi-Fi Protected Access (WPA) [21]. It was still based on the RC4 cipher and could be implemented through simple firmware upgrades to existing IEEE 802.11 devices. The main change to the WEP algorithm was that the inputs for the pseudorandom generator were less predictable; the Initialization Vector concept remained intact, but the shared key also changed for each packet, as described in the Temporal Key Integrity Protocol (TKIP). The final version of the standard implemented a new protocol, Wi-Fi Protected Access 2 (WPA2). No longer based on RC4, it replaced TKIP with the CCMP[2] protocol, based on the Advanced Encryption Standard (AES) algorithm. In most cases, new hardware was required for the new algorithm; the 2007 IEEE 802.11 standard included these amendments, and support for the newer algorithm was mandatory for Wi-Fi certification. Wi-Fi Protected Setup (WPS) was later introduced by the Wi-Fi Alliance[3], as an optional feature. It allows easier setup of a secure connection; instead of requiring long password inputs on every device, it allows negotiation of keys through the push of a button on the AP, or using a smaller PIN code.

All methods referred for IEEE 802.11 networks are based on a shared key that must be distributed through all the devices. This is most common in personal networks; corporate setups are usually combined with the IEEE 802.1X [22] standard. It defines the encapsulation of the Extensible Authentication Protocol (EAP) [23] over local and metropolitan networks. This protocol requires a dedicated Authentication, Authorization and Accounting (AAA)

[1]aircrack suite, `http://www.aircrack-ng.org/`
[2]Counter Cipher Mode with Block Chaining Message Authentication Code Protocol
[3]Wi-Fi alliance, `http://www.wi-fi.org/`

server such as RADIUS or Diameter. EAP is just a framework that specifies the transport of keying material, and supports many authentication methods. There are methods based on simple identity/password pairs, while more complex methods provide mutual identification through the use of Public Key Infrastructure (PKI) tokens. Some methods are based on SIM cards and other hardware-based mechanisms.

2.3 Internet Protocol

The first major version of the Internet Protocol was IP version 4 (IPv4) [24]. IP version 6 (IPv6) [25] is the widely supported version for replacing IPv4, with the most prominent change being the addressing space, now using 128 bit instead of the previous 32. IP addresses can be manually or dynamically allocated to devices. Some classes of addresses are intended for global connectivity, and are uniquely assigned to a particular device; others are limited to communication inside local networks (subnets). Two different subnets may allocate the same range of addresses to their devices without collision. Devices without global addresses are able to reach devices outside that scope through Network Address Translation (NAT) [26]. The Internet Control Message Protocol (ICMP) [27, 28] is used by devices for diagnostic and control purposes at the network layer. For example, an ICMP Echo Request is a message used to test whether a device exists with a given IP address.

Other than manually assigning IP addresses, there are protocols for automatic, dynamic attribution, like the DHCP [29, 30]. This is an application layer protocol; despite dealing with network layer configuration, it works by broadcasting User Datagram Protocol (UDP) packets over the network using a null source IP address and special UDP port reserved for the protocol. DHCP servers listen to these messages and respond by allocating (leasing) an unused IP address to the host. Leases expire after a period of time; if the host is still in the network, it is expected to attempt and renew the lease before it expires. After a lease expires, or if it is explicitly released, the DHCP server considers the IP address free for leasing to another host.

The IPv6 protocol defines an additional method for devices to configure themselves independently. It is called stateless autoconfiguration [31], in contrast with the “stateful” DHCP version 6 (DHCPv6) method, which depends on information provided by a dedicated server, that keeps track of the addresses for each device. The stateless mechanism allows a host to generate its own addresses using a combination of locally available information and information advertised by routers. Routers advertise prefixes that identify the subnet(s) associated with a link, while hosts generate an "interface identifier" that uniquely identifies an interface on a subnet (usually based on the link layer address). An address is formed by combining the two. In the absence of routers, a host can only generate link-local addresses. However, link-local addresses suffice for communication among nodes attached to the same link.

Other than IP addresses, DHCP servers provide additional bootstrap configurations including, among many others: static network routes, Network Time Protocol (NTP) servers and Domain Name System (DNS) servers. The DNS [32, 33] protocol is used to translate domain names (e.g. www.ua.pt) into IP addresses. This allows humans to reach servers without knowing their IP addresses, which are much harder to remember.

Chapter 3

GNU/Linux Network Management

Modern Central Processing Units (CPUs) define several privilege levels for executed code. The Intel[1] 64 and IA-32 Architecture [34] processors, for example, support 4 levels, with level 0 being the most restrictive, and 3 the least privileged, as depicted in Figure 3.1. Typically, level 0 corresponds to OS kernel code. In Linux this includes the typical memory management and input/output, but also driver modules, since they are compiled directly or as modules to the kernel. This is the only level at which software directly controls hardware; code running without such privileges is said to run in userspace. Linux considers only these two levels.

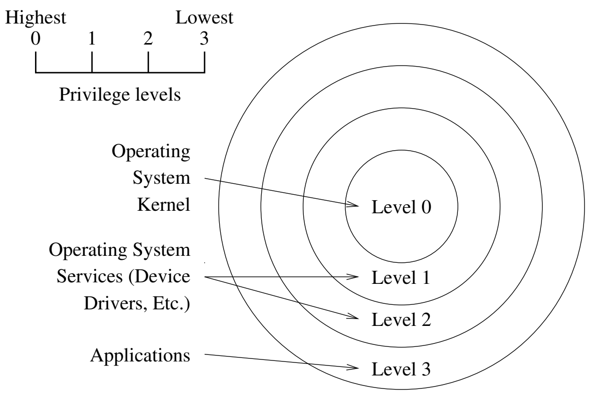


Figure 3.1: Intel protection rings.

Network Managing is a process that occurs at high level, including user interaction, but it implies hardware management as well. Also, many user applications might need to interact with the Network Manager itself in order to gather or control network information. The processes by which applications pass information between each other is called Inter-Process Communication (IPC).

To interact with the Operating System and hardware, programs need to have the kernel carry out operations for which they do not have the required privileges (manipulation of hardware and the kernel itself). Such mechanisms are usually referred to as system calls, or *syscalls*, for short.

[1]Intel, `http://www.intel.com/`

Section 3.1 will start by discussing system calls and other derived kernel interfaces, followed by an overview of IPC frameworks in Section 3.2. Sections 3.3 and 3.4 will present the most common tools and full-featured solutions for Network Managing, respectively.

3.1 Kernel/Hardware interfacing

The Linux kernel provides a set of different syscalls for different, specific operations. Two complementary examples are the `open()` and `close()` syscalls. A call to `open()` provides the user with a file descriptor for a file that either exists or is to be created by the operation. This file descriptor is the information that the process holds and must use for subsequent operations on the file, including writing (`write()`), reading (`read()`) and closing (`close()`).

Other than file management, the Linux kernel provides system calls for process and memory control (`execl()`, `exit()`, ...), system maintenance (`gettimeofday()`, `settimeofday()`, ...) and communication (`send()`, `recv()`, ...), among other standard services. However, a limited set of syscalls will never suffice for interaction with all existing and newly created hardware devices. Each system call must be officially assigned a unique number, and and its behavior and interface must not change through time. To avoid this problem in providing new device interfaces, a special syscall was created for control of non-standard hardware – `ioctl()`.

3.1.1 ioctl

ioctl is an abbreviation of input/output control, and provides an interface for operations on special files. Unlike common system calls, it can be called with a different number and type of parameters, depending on the intended destination and action to carry out. The call has the following signature:

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

To enact a generic interface, the method takes as parameters: a file descriptor, a request code number, and an optional pointer to arbitrary data for extra parameters[1]. The file descriptor identifies the destination itself. The request code number is an identifier, specified by the device driver, to indicate the action that is to be executed. Depending on the action, the additional parameter may refer to input arguments, output values, or both; this parameter was traditionally identified as `char *argp`, but the ellipsis were introduced later, to prevent type checking during compilation.

ioctl does what is called "syscall multiplexing", since it "does wildly different things depending on a flag argument" [35]. This presents various problems; code with several ioctl calls will easily become hard to read, since the meaning of each invocation is concealed in request code numbers and parameter structures. Wrapping the invocations may overcome human

[1]Note the deliberately expressed "optional" as, in this case, the ellipsis do not refer to a variable number of arguments, but to a single optional one.

readability problems, but the underlying complexity remains, and debugging will be difficult as invalid arguments to the function will not trigger compiler errors or even warnings. All of this does not endorse kernel design and security goals [36]; third party and proprietary device drivers exposing their interface through ioctl calls may not be audited for flaws or vulnerabilities, posing a great threat to system security and stability. This sorts of reasons caused ioctl to fall out of favor among kernel developers, and *sysfs* became one of the alternatives.

3.1.2 sysfs

The Linux kernel supports device hotplugging, and is permanently alert to hardware changes in the system. This information collected by the kernel is transformed into an object-oriented representation of the system, the device model, structured similarly to the example in Figure 3.2. The current device model design dates back to the Linux's 2.5 development cycle, and it is exposed to the userspace as a memory-based virtual filesystem, *sysfs*, mounted at the `/sys` directory.

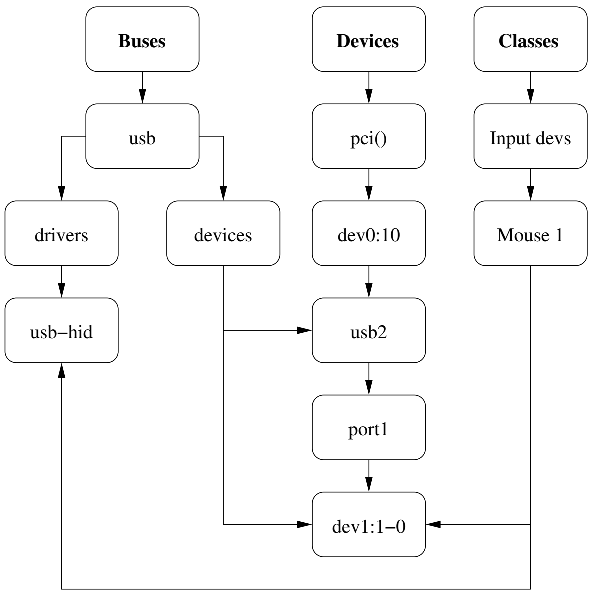


Figure 3.2: Linux device model.

A fundamental structure of the device model is the *kobject*. Just as the name points out, it is the base representation for a kernel object. When exported to the userspace, every item in the sysfs file tree has an underlying *kobject* that binds the structures to the kernel. Any direct manipulation of a kobject is automatically updated in the sysfs tree. sysfs entries for kobjects are always directories. Each kobject also exports one or more *attributes*. An attribute has a name (key), value, and an associated module responsible for its implementation. Each attribute results in a sysfs file with the same name; reads and writes to the file translate in calls to the *show* and *store* methods of its module, respectively. From a higher level perspective, directories become objects, files become attribute keys, and file contents become attribute values. So, for example, `/sys/class/net/wlan0/address` contains `00:11:22:33:44:55`, which is the value of the attribute `address` from the object `wlan0`.

Although it was initially to facilitate debugging, sysfs (formerly *debugfs*) became a replacement for ioctl [35]. There are limitations to this: sysfs conventions call for all attributes to contain a single value in a human readable text format, which makes the mechanism unsuitable for large data transfers, while also being limited to text-based interfaces. Moreover, the model does not directly support event signaling for userspace applications, and notifications have to be handled through separate frameworks. A different solution to replace ioctl was accepted in early Linux 2.0, based on sockets, called *Netlink*.

3.1.3 Netlink sockets

There is a large number of socket implementations, with varying types and domain scopes (or address families). Every socket implementation provides at least two types: stream and datagram [37]. Stream sockets operate in connected pairs and provide a reliable, bidirectional, byte-stream communication channel. Datagram sockets are connectionless and transfer data in separate messages, called datagrams, without any guarantee of delivery.

A socket domain specifies how a socket is addressed, as well as the scope of the communication. Common domains include `AF_UNIX`, which uses file system paths for addressing within the same machine, and `AF_INET/AF_INET6` for communication over the IPv4 and IPv6 Internet domains, respectively. In the Internet domain, datagram sockets employ the User Datagram Protocol and stream sockets employ the Transmission Control Protocol.

A socket is created with the `socket()` syscall. The `bind()`, method assigns it an address and the methods `send()` and `recv()` are then used to exchange information:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Netlink [38] is a datagram-oriented message system[1]. It allows message passing between the kernel and userspace, but also among user-space processes, for Inter-Process Communication. Netlink sockets are addressed in the `AF_NETLINK` domain, which is similar to `AF_UNIX`, but depends on Process IDs (PIDs) for addressing rather than file system paths. The protocol messages consist of a header, shown in Figure 3.3, and the additional payload attached to it. Large messages may be split into multiple datagrams; message ordering is not guaranteed, but the Sequence Number and Flags fields allow correct reconstruction of the payload at the destination.

Each Netlink protocol (family) defines their own message format to fill the payload, usually employing a stream of Type-Length-Value (TLV) attributes. The TLV format allows

[1]Netlink sockets support both `SOCK_RAW` (for raw network protocol access) and `SOCK_DGRAM` types, but no distinction is made between the two.

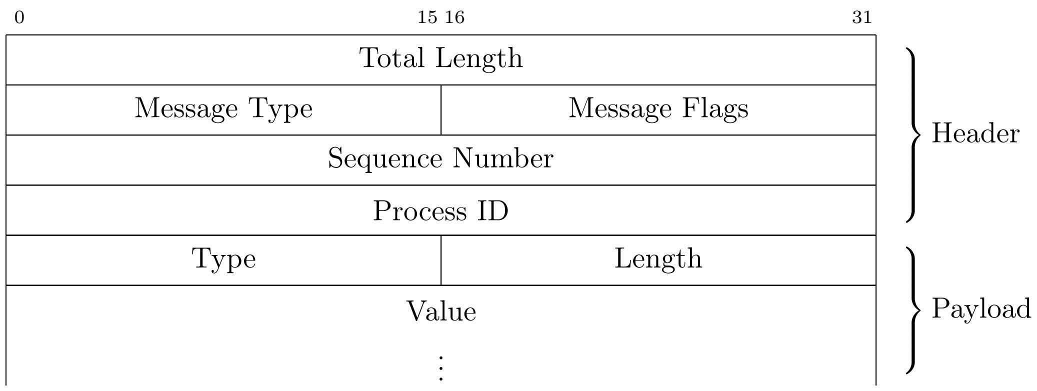


Figure 3.3: Netlink message format.

the addition of new attributes without breaking backward compatibility of existing applications [39]. Extending a protocol simply requires adding new attributes and updating the user-space application to support it; applications that do not support an attribute skip it, and the rest of the implementation remains intact[1]. This results in higher flexibility than either ioctl or sysfs solutions.

There are two communication types for a Netlink bus: unicast and multicast. Unicast is used for one to one communication; typically to send commands or queries from user to kernel-space, and receive their corresponding response. Multicast is a one to many communication channel, and is typically used by the kernel for event notifications; in fact, device model events, referred in Section 3.1.2, are propagated through Netlink.

Netlink supports up to 32 buses[2] in kernel space, each attached to one kernel subsystem, although several subsystems could share the same bus [39]. As of writing, this listing is supported:

- NETLINK_ROUTE can be used to modify routing tables and IP addresses both for IPv4 and IPv6, as well as changing link attributes, queuing disciplines, packet classifiers, etc. It is also used for broadcasting routing and link updates.
- NETLINK_FIREWALL and NETLINK_IP6_FW are used to transfer packets from netfilter, the Linux's packet filtering framework, to the userspace.
- NETLINK_NETFILTER serves as the interface for the netfilter framework, for packet mangling across each specific protocol stack.
- NETLINK_XFRM, for communication with the kernel module that handles Security Associations and Security Policies for Internet Protocol Security (IPsec) in Linux.
- NETLINK_USERSOCK, for user space socket protocols.
- NETLINK_W1, for the 1-wire subsystem.

[1]Removing or changing attributes will break implementations that depend on them.
[2]This limit is enforced for performance reasons, the protocol can handle 256 buses.

- NETLINK_ISCSI, for Open-iSCSI.
- NETLINK_INET_DIAG, for INET socket monitoring.
- NETLINK_NFLOG, for Netfilter/iptables ULOG.
- NETLINK_SELINUX, for SELinux event notifications.
- NETLINK_AUDIT, for auditing, as a complement to SELinux.
- NETLINK_FIB_LOOKUP, to lookup Forward Information Base entries.
- NETLINK_CONNECTOR, for the special kernel connector driver.
- NETLINK_DNRTMSG, for DECnet routing messages.
- NETLINK_KOBJECT_UEVENT, for device model notifications to the userspace.

Given the concern that the number of Netlink family numbers might exhaust in the near future, an additional family was created: NETLINK_GENERIC. Generic Netlink acts as a Netlink multiplexer, allowing multiple interfaces on a single Netlink bus. Family IDs for new interfaces are created and discovered at runtime, based on text names. Generic Netlink are encapsulated in the Netlink payload as shown in Figure 3.4.

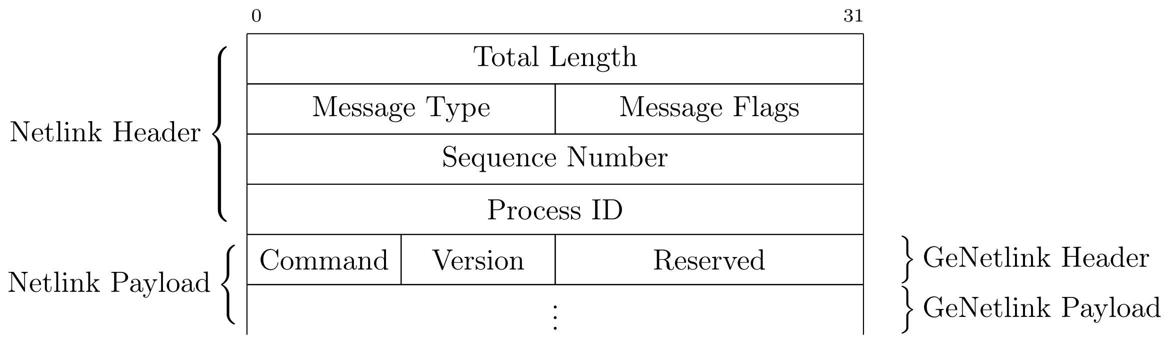


Figure 3.4: Generic Netlink Message format.

For network managing purposes, Route Netlink (NETLINK_ROUTE) and a specific interface on Generic Netlink, nl80211, are of great importance, and will be now further extended.

3.1.3.1 Route Netlink

Route Netlink is considered the most mature of all the Netlink family protocols [40]. The protocol itself can be subdivided into classes, with specific messages, each of which interfacing with a specific part of the Linux kernel's network routing system in the following list:

- **LINKS**: create, remove or get information about a specific network interface.
- **ADDRESSES**: add, remove or receive information about an address associated to an interface.

- **ROUTES**: create, remove or receive information about a network route.
- **NEIGHBORS**: add, remove or receive information about a neighbor table entry.
- **RULES**: add, delete or retrieve a routing rule.
- **DISCIPLINES**: add, remove or get a queuing discipline.
- **CLASSES**: add, remove or get a traffic class.
- **FILTERS**: add, remove or receive information about a traffic filter.

Each class defines an ancillary data type for the messages, and is optionally followed by a set of subsequent attributes of varying length. These data types define the operations on a set of objects that can be represented as a table. There are three primitives for operation on those tables:

- **GET**, to retrieve entries in the table.
- **NEW**, to create or edit table entries.
- **DEL**, to delete table entries.

For the GET message, most of these fields may be set to an unspecified value, to act like wildcards, or they can be specified to filter results. NEW and DEL messages may be used as commands, but are also commonly available as multicast messages for notifications on certain groups.

For the scope of this dissertation, the purpose of the LINK, ADDR and ROUTE message classes are particularly relevant, and will be presently explained.

The LINK messages

The LINK family of messages allows a user of the Route Netlink protocol to manipulate information about network interfaces on the system. Each underlying entry on the LINK table refers to a network interface that exists on the computer, whether physical or virtual. The three LINK message types are:

- **RTM_NEWLINK**: create or edit network interfaces.
- **RTM_DELLINK**: destroy network interfaces.
- **RTM_GETLINK**: retrieve information about network interfaces.

The NEW and DEL messages may be transmitted as notifications on the RTMGRP_LINK multicast group of the Route protocol; the NEW message indicates a link creation or link parameter changes, and the DEL message indicates a link removal. The ancillary data type for the LINK messages is defined as follows:

```
#include <linux/rtnetlink.h>

struct ifinfomsg {
    unsigned char ifi_family;
    unsigned short ifi_type;
    int ifi_index;
    unsigned int ifi_flags;
    unsigned int ifi_change;
};
```

- **ifi_family**: set to AF_UNSPEC, except for interfaces with associated IPv6 addresses, in which case the field is AF_INET6.
- **ifi_type**: the interface type, supporting various wired and wireless technologies.
- **ifi_index**: unique identifier of the interface in the system. It is not statically associated with the hardware and may vary at each system boot or configuration changes.
- **ifi_flags**: the interface flags, used to define an interface's power state, for example.
- **ifi_change**: reserved for future use, currently set to `0xFFFFFFFF`.

In each message, this structure should be followed by additional attributes, including the interface's physical address, its MTU the label of the interface in the system, statistics data, etc.

The ADDR messages

Route Netlink ADDR messages refer to the manipulation of IP addresses in the network interfaces. It is composed as well by a group of three message types:

- **RTM_NEWADDR**: add or edit interface addresses.
- **RTM_DELADDR**: remove addresses from an interface.
- **RTM_GETADDR**: get addresses from interfaces.

These messages support IPv4 and IPv6 addresses, and an interface can be assigned multiple IP addresses (previously this was only achieved with alias devices). Much like in the LINK messages scenario, it is possible to subscribe to a multicast channel that will generate RTM_NEWADDR and RTM_DELADDR to announce changes to the address table of each network interface. Every ADDR message contains the following structure:

```
#include <linux/rtnetlink.h>

struct ifaddrmsg {
    unsigned char ifa_family;
    unsigned char ifa_prefixlen;
    unsigned char ifa_flags;
    unsigned char ifa_scope;
    int ifa_index;
};
```

- **ifa_family**: the address family to which this address belongs. This is currently limited to AF_INET and AF_INET6.
- **ifa_prefixlen**: the length of the address mask of this address.
- **ifa_flags**: properties of this address. Whether this is a primary or secondary address for the interface, permanent or temporary, etc.
- **ifa_scope**: the scope of the address, such as link-local (RT_SCOPE_LINK), site-local (RT_SCOPE_SITE) or global (RT_SCOPE_UNIVERSE). Two additional scopes are available, for addresses limited to host boundaries (RT_SCOPE_HOST) and a special case "nowhere" (RT_SCOPE_NOWHERE).
- **ifa_ifindex**: the interface this address is associated with. This may be regarded as a foreign key to the LINK table, as this value uniquely identifies a LINK object.

ADDR messages may also carry additional attributes regarding the address itself (such as the associated local, broadcast and anycast addresses), or the interface it is associated with (its interface name or mac address).

The ROUTE messages

These messages baptize the protocol name and, as it points out, they refer to IP routing table management. Without exception, there are three message types:

- **RTM_NEWROUTE**: create or edit routes.
- **RTM_DELROUTE**: remove routes or notify of route removal.
- **RTM_GETROUTE**: get existing configured routes.

The underlying data type is the `rtmsg`, declared as follows:

```
#include <linux/rtnetlink.h>

struct rtmsg {
    unsigned char rtm_family;
    unsigned char rtm_dst_len
    unsigned char rtm_src_len;
    unsigned char rtm_tos;
    unsigned char rtm_table;
    unsigned char rtm_protocol;
    unsigned char rtm_scope;
    unsigned char rtm_type;
    unsigned int rtm_flags;
};
```

- **rtm_family**: similar to the ifa_family field in ADDR messages.
- **rtm_dst_len**: the length of the destination address of the route entry, included in the additional message attributes.
- **rtm_src_len**: the length of the source address of the route entry, included in the additional message attributes.
- **rtm_tos**: the Type of Service (TOS) indicator for the route. This makes use of the TOS header field in the IPv4 header, replaced by the traffic class field in IPv6.
- **rtm_table**: specifies the routing table ID. Can be default (RT_TABLE_DEFAULT), main (RT_TABLE_MAIN) or local (RT_TABLE_LOCAL).
- **rtm_protocol**: how the route originated. Possible values are RTPROT_KERNEL, if originated by the kernel; RTPROT_REDIRECT if originated by an ICMP redirect; RTPROT_BOOT during boot and RTPROT_STATIC if set by the administrator.
- **rtm_scope**: pertains to the scope of the destination, with values ranging similarly to the `ifa_scope` field of ADDR messages.
- **rtm_type**: the type of this route. Possible values include RTN_UNREACHABLE, RTN_UNICAST, RTN_LOCAL, RTN_BROADCAST and RTN_ANYCAST.
- **rtm_flags**: various extra parameters for a route.

Just like with other messages, ROUTE messages may require several additional attributes, such as:

- **RTA_DST**: the route destination address.
- **RTA_SRC**: the route source address.
- **RTA_IIF**: the input interface index.
- **RTA_OIF**: the output interface index.
- **RTA_GATEWAY**: the default gateway of the route.
- **RTA_PRIORITY**: the priority of this route.
- **RTA_PREFSRC**: the preferred source address for an IPv6 route.
- **RTA_METRICS**: the route cost metric.

Summary

The Linux network subsystem exposed by the Route Netlink protocol is significantly larger than the subset presented in this subsection. As mentioned before, it also allows the configuration of neighbor (Address Resolution Protocol (ARP)) tables, routing rules, traffic classes and filters, and queuing disciplines. These fall out of basic network managing processes and are not further documented here. For an extensive analysis of this protocol, section 7 of the "rtnetlink" GNU/Linux manpage can be consulted.

3.1.3.2 nl80211

Several aspects of a network interface can be manipulated by the Route Netlink kernel interface, but there is a very wide range of variables to take into account for an IEEE 802.11 interface that the routing subsystem is not aware of. In essence, there is a whole different kind of interface that the kernel needs to expose to the userspace in order to support wireless network management.

Traditionally, the WLAN Application Programming Interface (API) for Linux was Wireless Extensions (WEs)[1], developed by Jean Tourrilhes and sponsored by Hewlett Packard[2]. This framework is still used, but it is based on *ioctl* calls, so it is being deprecated in light of the advantages of Netlink. nl80211 operates on the Generic Netlink bus and since its beginning in 2006, together with cfg80211, aims at replacing the WEs [41]. It is still a work in progress; current IEEE 802.11 drivers still support the WEs, but it is already the recommended architecture for new applications being developed. Interpreting Figure 3.5, the real interface for IEEE 802.11 drivers is cfg80211, and nl80211 are just different userspace endpoints for that interface, both for Station and AP applications.

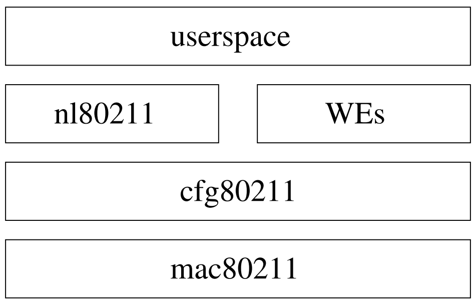


Figure 3.5: Linux Wireless layered interface.

Unlike the Route Netlink protocol, the interface fails to follow a strict model of categorized messages, and instead relies on a series of disparate commands and parameters to carry out a large array of operations, which can be roughly grouped in the categories of the following sections.

[1]Wireless Extensions, `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html`
[2]Hewlett Packard, `http://www.hp.com/`

Wireless interface management

nl80211 allows getting and setting many interface attributes, and handling virtual interfaces. There are four CMD_<OP>_WIPHY and CMD_<OP>_INTERFACE, where <OP> stands for one of GET, SET, NEW or DEL.

WIPHY objects correspond to a physical wireless device in the system. Physical devices are not created or deleted from userspace, this takes place at kernel level, according to supported devices and device drivers. The DEL_WIPHY command is used as a notification that a physical IEEE 802.11 device was removed from the device model[1]; the NEW_WIPHY command is used as a notification that a new physical device is available, as a notification of a rename, or as a response to a GET_WIPHY request. With the GET_WIPHY request one can retrieve a list of available devices, or a very extensive list of hardware attributes and capabilities (supported frequency bands, authentication mechanisms, etc), as well as a list of accepted commands and interface modes (Infrastructure, IBSS, etc). The SET_WIPHY command allows the configuration of transmission power, transmission queue parameters, antenna selection and some threshold and retry parameters.

Each WIPHY can hold many virtual interfaces, created with the NEW_INTERFACE command. Virtual interfaces define roles for the underlying WIPHY (Station, AP, Mesh Point, etc). Usually there is a single virtual interface on top of the physical hardware due to the fact that not all devices are able to operate with virtual MAC addresses, and also because concurrent operating modes may face a number of restrictions. Device drivers commonly define a default INTERFACE for a WIPHY object and allow further modifications, if they support other operation modes (but support for changing modes at runtime is not mandatory). The NEW command may also be used to notify of newly created virtual interfaces, and as a reply to GET_INTERFACE requests for, retrieving an interface's configuration. Finally, SET_INTERFACE is used to change the type of an existing interface.

Power management

Other than ON/OFF, the IEEE 802.11 interfaces implement power options described by the standard, such as letting the device sleep between beacons from the AP. This behavior can be enabled or disabled through the SET_POWER_SAVE command. Checking the current configuration is done with the GET_POWER_SAVE command.

The Linux kernel also supports various enablers for the non-standard "Wake on WLAN" feature. This feature is supported while connected to a network, based on triggers like the reception of a user specified pattern, a magic packet [42], or disconnection from an AP. While disconnected from a network, a trigger such as the discovery of a new network can be used. The list of supported triggers for each device is available in the WIPHY properties and through the GET_WOWLAN command. To activate a trigger, the SET_WOWLAN command is used.

[1]This could be either because the network hardware is physically removed or simple because the driver module was removed from the running kernel.

Regulatory enforcement

Though IEEE 802.11 is a globally accepted standard for wireless connectivity, each government claims the right to regulate the usage of the electromagnetic spectrum of their territory. Wireless devices are sold throughout the planet and are usually hardware-enabled for global operation. Some devices support programming of the regulatory permissions to an Electrically Erasable Programmable Read-Only Memory (EEPROM), but the regulations must be enforced by software. The Linux developers respect this authority and provide the means to prevent accidental defiance of regulations. Moreover, public sources of the Linux kernel will never provide options to disable regulatory enforcement.

Linux relies on a regulatory database built and maintained by John Linville, which he signs with his private RSA key [43]; this database is publicly available, along with the associated public key. A userspace tool, by the name Central Regulatory Domain Agent (CRDA), checks the signature against a list of public keys built into the execution binary or in a preconfigured directory, and uploads the regulatory domains into the kernel.

The Linux desktop is responsible for determining the user location, through services like *GeoClue*[1], and make use of the SET_REG command to set the proper regulatory domain for the specific machine. The GET_REG command gets the current configured domain. A REG_CHANGE message informs the userspace processes that the domain has been changed, providing details of the change request, such as the request author.

When CRDA is installed but no domain is configured, the kernel allows passive scanning to occur in what is referred to as "world roam". If, during a passive scan, the kernel detects an AP on a channel that is not globally allowed, it will signal userspace programs with a REG_BEACON_HINT message and will allow active scanning on that band. This does not limit the behavior in channels 1 through 11 on the $2.4Ghz$ band, as they are always enabled worldwide.

Scanning

nl80211 supports triggering scans at any moment as well as the configuration of scheduled scans at regular intervals. With the TRIGGER_SCAN command a scan is immediately issued on all channels, depending on the current regulatory domain. Optionally, a list of frequencies may be given, as well as a list of SSIDs and extra Information Elements (IEs). Scans are passive by default; active scans are performed when a list of SSIDs is given. If the user application wants to request an active scan on all channels, but does not wish to specify an SSID, an empty string must be used (acting as a wildcard). Scheduled scans are configured with START_SCHED_SCAN, specifying the scan cycle interval, in milliseconds. Scan results are returned either with a NEW_SCAN_RESULTS or SCHED_SCAN_RESULTS message, depending on the type of scan performed. There is a SCHED_SCAN_STOPPED message to inform of the cancellation of a configured scheduled scan, with the STOP_SCHED_SCAN

[1] *GeoClue*, `http://www.freedesktop.org/wiki/Software/GeoClue`

command. A SCAN_ABORTED message may be issued if a scan is interruped, possibly carrying partial scan results only.

Association and authentication actions

Joining a BSS consists on the authentication to the AP, followed by the association operation. The Linux implementation of IEEE 802.11 supports the following authentication mechanisms:

- **OPEN_SYSTEM**: used for unprotected networks, and the first step of WPA/WPA2 protected networks. WEP also supports this authentication procedure.
- **SHARED_KEY**: used for WEP authentication only.
- **NETWORK_EAP**: for Lightweight Extensible Authentication Protocol (LEAP) protected networks.
- **FT**: for Fast BSS Transition, defined in the IEEE 802.11r [44] amendment for the Wi-Fi standard, providing mechanisms that minimize losses during a BSS transition.

These authentication types are used with the AUTHENTICATE command. After successful authentication, an association may be attempted with the ASSOCIATE command. An additional command, CONNECT, merges the two steps in a single command. Once connected, the DISASSOCIATE, DEAUTHENTICATE and DISCONNECT commands perform the opposite operation. Upon completion, all of these commands generate events of the same message type, carrying an attribute that indicates the success status of the corresponding action.

Frame transmission and registration support is also in place, for userpace entities to interact with frames not to be handled by the kernel. IEEE 802.1X userspace daemons are implemented using this feature. The REGISTER_FRAME (or the REGISTER_ACTION alias) command allows the registration for receiving frames based on a match attribute for the first few bytes on the frame. Registrations are only dropped once the Netlink socket is closed. FRAME messages are used as commands and notifications for frame transmission and reception, respectively.

Notification groups

Most of the mentioned functionalities refer to direct control of the interface, but some event messages are also available. Userspace processes that do not necessarily want to control an interface may be interested in certain events. Most events were already mentioned, but a complete overview is given here for the four available event notification groups:

- **Config**: this group is used for notifications of a WIPHY change, like a device rename.
- **MLME**: the group's name stands for MAC Sublayer Management Entity. It refers to authentication and association management, and other link layer aspects of the IEEE 802.11 wireless standard. The following list contains some events available in this group:

  - **Connect**, **Disconnect**: the list of of messages pertaining to authentication and association are broadcasted in this group. Each message will contain an attribute indicating a failure status, and a code indicating the cause for the event.
  - **Probe Status**: every time a probe request frame is sent in search of another station, the response is broadcasted to the userspace with the PROBE_CLIENT message, containing the other station's address, and whether it ACKnowledged the probe request.
  - **Connection Quality Monitor (CQM) notification**: users are able to configure triggers to monitor signal strength levels and transmission rates. The NOTIFY_CQM message is used to notify that a certain threshold was triggered.
  - **PMKSA candidate**: the PMKSA_CANDIDATE message is used to notify that a given AP matches the Pairwise Master Key (PMK) that the station holds.
  - **Roam**: a ROAM message indicates that the Station roamed to a new AP.
  - **Station add/delete**: the NEW_STATION and DEL_STATION commands indicate that a peer was removed from the known list.

- **Regulatory**: a regulatory change is announced through the REG_CHANGE event, and the REG_BEACON_HINT message indicates that other 802.11 devices were detected operating in frequencies outside the configured regulatory domain.
- **Scan**: the events in this group indicate a scan start (TRIGGER_SCAN), or a scan interruption (SCAN_ABORTED). Regular scan completions are announced with the NEW_SCAN_RESULTS message; scheduled scans with SCHED_SCAN_RESULTS.

Summary

Much of the described functionality is documented in the official Linux Wireless website[1], which includes the kernel source documentation of the nl80211 implementation. However, certain mentioned aspects are not detailed in those documents, and are based on an analysis of the current nl80211 source code. The documentation suffices for most uses, although more advanced users will look into the kernel source code to be certain of some specific features, and what commands will translate to. At the time of writing the interface is also under development, and new features are still being added, so older kernel versions might affect user applications.

3.2 Application interfacing

IPC need not always occur between vertical layers (i.e. multiple interfaces that translate higher level operations to lower level hardware control, and *vice versa*). Higher level programs communicate with each other at the same level (horizontally). In this context, take as an

[1]Linux Wireless, `http://wireless.kernel.org/`

example the case of an application that needs network connectivity, like a Web Browser or Instant Messenger: instead of blindly attempting to open a socket and connect to a server, these applications could benefit from an interface that would allow them to known whether the system is connected to the Internet. This kind of interface would not require kernel communication, since the Network Manager itself is also a user application, and it takes the burden of communicating with the kernel for network specific operations. As another example, an Instant Messenger application alone would be able to notify the user of incoming messages, but most systems contain specific software for user notifications; this notification software needs to provide an interface so that other programs use it to present notifications to the users.

According to the examples, the Instant Messenger is always a client of other programs, but it could also play a serving role: for example, it could provide e-mail clients an interface for checking whether a person is online before composing a message, and thus permitting it to suggest a real-time conversation instead of e-mail exchanges.

Recalling the beginning of the chapter, userspace-kernel communication faces the problem of different running privileges. In modern OSs, not only the kernel code runs in a separate space from the users', but also each user program is allocated its own virtual memory space. This is a basic security measure that otherwise hinders communication between programs; the kernel must then provide means to circumvent this restriction. One has already been mentioned, which is Netlink. However, despite the flexibility of the protocol, it is a fact that its usage is mostly for kernel-userspace communication.

Even before Netlink existed, there were other basic IPC solutions, which lost preference over time as more complex applications appeared. Common IPC solutions include the following list of concepts [37]:

- **Shared Memory**: based on requesting a kernel for a shared memory segment that can be attached to multiple user processes transparently.
- **Pipes**: provide easy mechanisms to stream data from one process (producer) to another (consumer).
- **FIFOs**: similar to Pipes, but allowing multiple producers and consumers.
- **Message Queues**: similar to the Pipe and FIFO concepts, but preserving data boundaries, and allowing setting priority parameters for data for extraction.
- **Semaphores**: a basic mechanism of synchronizing multiple programs, often needed for controlling access by multiple processes to a common resource.

These concepts are only devised to solve the problem of program communication inside the same host. However, it is common to use network sockets for communication in the same host, enabling communication between different hosts with little or no modification to the software.

3.2.1 Networking Sockets

The first paragraphs of Section 3.1.3 explain the basics of the socket mechanism. Sockets provide a far more flexible solution than shared memory, for example, especially considering they allow communication among processes not only on the same machine, but also on different machines, over the network. FIFOs are also not capable of communicating over the network but even locally they are less performing than UNIX domain sockets [45].

Most complex application interfaces in GNU/Linux are built on top of sockets. Some programs implement interfaces directly on top of sockets, while others use more complex frameworks implementing higher-level interfaces based on the object-oriented paradigm, for example. Many large projects implement their own framework; the Internet Systems Consortium (ISC)[1] DHCP server has a framework by the name OMAPI. The X Windows System has its own socket based protocol. However, there are various frameworks that facilitate Inter-Process Communication, such as CORBA and D-Bus.

3.2.2 CORBA

The Common Object Request Broker Architecture (CORBA)[2] is a standard for communication between software components using object-oriented models. It allows developers to seamlessly combine programs written in different languages or across different platforms by mitigating low level communication and programming problems such as byte order, the packing and unpacking of complex structures, etc. It is widely supported across Operating Systems and programming languages, with standard mappings for C, C++, Java, and others.

Version 1.0 of the CORBA specification was released in 1991. Currently, it is still subject to various studies [46, 47] and use-case scenarios [48, 49, 50]. However it has lost popularity in software deployment scenarios such as generic purpose OSs, due to its great complexity and lack of security features, among other reasons [51]. The Gnome[3] Desktop Environment, is an example of a project that dropped its CORBA-based framework (Bonobo), in favour a different solution; in this case, D-Bus.

3.2.3 D-Bus

Often regarded as an abbreviation of Desktop Bus, *D-Bus*[4] is an IPC system built on top of a one-to-one message passing framework. Its specification is currently defined by the freedesktop.org[5] project. The framework is intended for host-local communication, but there are also plans to support crossing host boundaries, using the TCP/IP stack. There are numerous D-Bus bindings for virtually every programming language (although some are still in development), including Ada, C, C++, Haskell, Java, Objective-C, Perl, Python, Scheme

[1]Internet Systems Consortium, `https://www.isc.org/`

[2]Common Object Request Broker Architecture, `http://www.omg.org/corba/`

[3]Gnome, `http://www.gnome.org/`

[4]D-Bus, `http://www.freedesktop.org/wiki/Software/dbus`

[5]freedesktop.org, `http://www.freedesktop.org/`

and Tcl. The Glib[1] and Qt[2] frameworks, on which many projects rely, provide easier bindings for D-Bus development.

Communication between applications can happen directly, with the D-Bus library, or through a message bus daemon, which is required for many-to-many communication. The daemon is a central server that: keeps track of applications that use the communication buses; routes messages between them; and launches new applications, if their services are requested. There are two bus types: session and system. One session bus is launched per user login, or desktop session, and is used for communication between applications in the user session. The system bus is intended for communication with applications common to multiple desktop sessions. A D-Bus interface for a Network Manager application would have to run on a system bus, since there is only one instance of networking hardware for multiple potential user sessions. An Instant Messenger's interface, however, would be contained in a single login session, since it would contain personal user information. Moreover, D-Bus is aware of user identities, and supports flexible authentication mechanisms and access controls between applications.

3.2.3.1 Connection

When connecting to a bus, an application may register one or more names to the connection. These are called *bus names* and are composed by keywords separated by dots, much like an Internet domain name: `org.freedesktop.NetworkManager`. This name must be well-known to the respective application interface, since it is used to address/discover the service. It is possible to have different connections with the same name, but the name is not shared. Instead, the D-Bus daemon stores a queue of connections that request the usage of each name. The first connection to get the name is the *primary owner*. A second connection will acquire the same name when the primary owner releases it, or terminates the connection. In addition to the requested bus names, the central daemon assigns each connection an immutable *unique connection name* starting with a colon (e.g. `:235-34`). This name is never reused on the same bus, even when the corresponding connection is closed and others are created.

3.2.3.2 Interface

A connection on a bus exports its services in any number of interfaces, or contracts, in the form of *object*s. An object is addressed by a Uniform Resource Locator (URL) path, e.g. `/org/freedesktop/NetworkManager/Devices/0`, and the interfaces it implements are identified similarly to a bus name: `org.freedesktop.NetworkManager.Device`. An object path may be arbitrary, but the interface identifier must be well-known. An interface defines the *properties*, *signals* and *methods*, that are available on an object, including their input and output parameters.

[1]Glib libraries, `http://www.gtk.org/`

[2]Qt libraries, `http://qt.nokia.com/`

- **Methods** are similar to C++ or Java class methods. A method may take input or output parameters, and return results. Output parameters are generally used when a method generates more than one result. When invoking a method causes an error, the requesting application receives an *exception* instead of the usual result.

- **Signals** can carry parameters, like methods, but are not invoked by clients. Instead, they are generated by the server object. The bus daemon then delivers the signal to every client of that object. This is similar to a multicast message, in that it is propagated only to certain clients on a bus.

- **Properties** resemble C++ or Java class attributes. However, they bear a simpler access restriction scheme: a property ha either *read*, *write* or *readwrite* permissions.

To access these members of an object, a client of an interface first acquires a *proxy* of that object. A proxy is a client-local representation of the object on the server that is really accessed through the bus. Most D-Bus bindings hide the details and coding of message marshalling. In fact, many languages make the manipulation of proxies, much like if the object was local to the same program. Multiple requests on a proxy are guaranteed to be carried out in the same order in the original object. The same is true for multiple replies to the same client. However, no ordering is guaranteed for multiple requests from different clients (or proxies on the same program, but using different connections). D-Bus also guarantees message delivery and no requests are dropped, unless ill-formed.

3.2.3.3 Type system

Data is passed among the processes in messages. Messages have a header and a body. The composition of the body payload is defined in the header, along with the sender, the destination, and other information. D-Bus has a type system that regulates how values of various data types can be serialized into a sequence of bytes for the message payload. Each data type has a type signature in the ASCII format. The 32-bit integer type is identified by the ASCII character "`i`". A body holding two integers has the "`ii`" header signature. There are more signature codes that allow encoding arrays, structs, strings, object paths, and even variants, which can hold any of the other types (for this data type, the signature is included in the payload). For example, a signature such as "`ybnqiuxtdsoa(vh)`" represents a block composed by the following sequence: `BYTE, BOOLEAN, INT16, UINT16, INT32, UINT32, INT64, UINT64, DOUBLE, STRING, OBJECT_PATH, ARRAY of STRUCT of (VARIANT, UNIX_FILE_DESCRIPTOR).`

3.2.3.4 Summary

K Desktop Environment (KDE)[1] had its own IPC framework, named Desktop Communication Protocol (DCOP), and Gnome also focused on their own communication framework (Bonobo). D-Bus does not have any Desktop Environment (DE) dependency; it was designed from

[1]KDE, `http://www.kde.org/`

scratch, albeit influenced by KDE's DCOP system [52]. KDE is currently using D-Bus, and Gnome has also abandoned its own bus messaging system in favor of the new framework. D-Bus is part of the cross-desktop project freedesktop.org, to which Red Hat[1] is the primary contributor, and it has become the desktop-agnostic IPC mechanism of choice [52].

3.3 Individual network management tools

Having covered the requirements for kernel and process communication, this section focuses on the analysis of various simple tools for managing the Linux network subsystem. All of them are typically developed for command line usage, but some expose programming interfaces and are suitable for usage by other programs.

3.3.1 Core tools

Although *iproute2*[2] exists since 2.2 Linux kernel, the latest major release of *net-tools*[3] (1.60) dates back to 2001, and is still available in most (if not all) distributions. *net-tools* is an ancient package, and many systems and administrators depended on it for a long time. Surprisingly, some still do. *net-tools* was mostly based on ioctl calls, and provided a series of executable commands for different tasks:

- **ifconfig**: configuration of layer 2 and 3 addresses and links, including power up/down.
- **route**: manipulation routing tables of the IP protocol.
- **arp**: manipulation of the kernel's IP network neighbour cache.
- **iptunnel**: management of IP encapsulation tunnels.
- **ipmaddr**: multicast address management.
- **netstat**: various network metrics and statistics.

One additional tool, `vconfig`, was not really part of the *net-tools* collection. It contained the code for the IEEE 802.1Q Virtual Local Area Network (VLAN) protocol implementation on Linux, but it was later added to the kernel, and its functions are now part of *iproute2*.

iproute2 provides the same features of the *net-tools* toolkit, the biggest difference to the user being the syntax. Internally, it uses the Netlink protocol instead of ioctl calls. Table 3.1 shows the *iproute2* commands that replace the old *net-tools* utilities.

Individually, these tools allow a user to power an interface on/up or off/down (e.g.: `ip link set dev eth0 down`), add an IP address to it (e.g.: `ip addr 10.10.10.9/30 dev eth0`), change the MAC address (e.g.: `ip link set dev eth0 address 00:11:22:33:45:55`), etc. Neither of these tools offers a programming interface of any sort; the software is just a

[1] Red Hat, `http://www.redhat.com/`
[2] *iproute2*, `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`
[3] *net-tools*, `http://sourceforge.net/projects/net-tools/`

net-tools	**iproute2**
`ifconfig`	`ip addr`
	`ip link`
`vconfig`	
`route`	`ip route`
`arp`	`ip neigh`
`iptunnel`	`ip tunnel`
`ipmaddr`	`ip maddr`
`netstat`	`ss`

Table 3.1: *net-tools* replacement by *iproute2*.

group of isolated executables that act on command line invocation, which limits usage for manual calls or scripted invocation. Also, the programs make no decision on their own; it is up to their user to do the right thing at the right time.

3.3.2 Wireless tools

An analogy of the *net-tools*/*iproute2* affinity can be made for the specific case of wireless device tools for GNU/Linux. The *wireless-tools*[1] project is part of the sponsorship from Hewlett Packard for the Wireless Extensions development (mentioned in Section 3.1.3.2). Sitting on top of the WEs, the *wireless-tools* depend on older ioctl calls, and only in certain cases benefits from the new Netlink protocol. Essentially, *wireless-tools* are marked for deprecation on the GNU/Linux ecosystem in a similar fashion to *net-tools*. *iw*[2] replaces the *wireless-tools* while also serving as a use-case for most of the nl80211 protocol features.

Just as with the *net-tools*/*iproute2*, both packages provide only command line interfaces for tasks specific to wireless devices. Table 3.2 contains a listing of those tasks and the difference between the wireless solutions. An additional tool, `ifrename`, is included in the *wireless-tools* package. It allows renaming network interfaces and, since it is not specific to wireless devices, it is not listed in the comparison table.

wireless-tools	**iw**	**description**
`iwconfig`	`iw get/set`	Manipulate basic wireless parameters
`iwevent`	`iw event`	Display wireless events
`iwgetid`	`iw link`	Get current network information
`iwlist`	`iw scan`	Initiate scanning
`iwspy`	`iw station`	Get node information
`iwpriv`	`--`	Allows setting driver-specific settings

Table 3.2: Comparison between *wireless-tools* and *iw* tools.

[1] *wireless-tools*, `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html`
[2] *iw*, `http://wireless.kernel.org/en/users/Documentation/iw`

3.3.3 Authentication supplicants

Open1X[1] is an attempt at the development of an open-source supplicant implementation for the IEEE 802.1X protocol. The latest update on the project was in January 2011, with an updated development version of the *XSupplicant* tool. It supports both Linux and Microsoft Windows Operating Systems, but many GNU/Linux distributions stopped providing support for the package.

wpa_supplicant[2] stemmed as an overlay for XSupplicant, providing WPA/WPA2 features that the former implementation lacked. Over time it separated completely from XSupplicant, and is currently actively developed for an increasing number of 802.1X EAP methods, with support for Ethernet networks (EAP over LAN (EAPoL)). Wi-Fi Protected Setup support is also in the feature list.

wpa_supplicant offers an autonomous daemon that accepts network configurations, based on which it will try to authenticate and associate to a network. It uses more than one means to communicate with the kernel by implementing various interface drivers. It usually relied on the Wireless Extensions API, but some Qualcomm Atheros[3] chipsets, for example, required a dedicated interface for the *madwifi* driver. More recently, the standard nl80211 interface unifies the access to all device driver functions.

The supplicant accepts a configuration file providing a series of blocks containing network parameters, similar to the following example:

```
network={
    ssid="eduroam"
    scan_ssid=1
    key_mgmt=WPA-EAP
    pairwise=CCMP TKIP
    group=CCMP TKIP
    eap=PEAP
    phase2="auth=MSCHAPV2"
    ca_cert="/etc/ssl/certs/ca-certificates.crt"
    identity="user@ua.pt"
    password="secret"
}
```

When running by itself, *wpa_supplicant* regularly scans for new networks, and automatically joins networks based on the order of the network blocks in the configuration file, network security level, and signal strength. However, it offers two interfaces for run-time configuration: a proprietary control library and a D-Bus interface.

The proprietary control library is developed with the C language, and requires linking with the provided objects; this restricts usage to C and C++ software, although other programming languages offer solutions for linkage with C libraries. It communicates with the daemon through Unix or UDP sockets, and even Pipes. This library provides an extensive list

[1] Open1X, `http://open1x.sourceforge.net/`

[2] *wpa_supplicant*, `http://hostap.epitest.fi/wpa_supplicant/`

[3] Qualcomm Atheros, `http://www.qca.qualcomm.com/`

of predefined commands accessible through a `wpa_ctrl_request` method, available for invocation after a `wpa_ctrl_attach` where the user specifies the interface to be managed. The commands are specified in text format, and replies are also text strings; for example, checking if the daemon is accessible requires sending the "`PING`" command and successfully parse a "`PONG`" response, otherwise *wpa_supplicant* is not available (the `wpa_crtl_attach` command is actually equivalent to a `wpa_ctrl_request` with the `''ATTACH''` command). Further commands include:

- `INTERFACES`: get the list of available interfaces.
- `ADD_NETWORK` and `REMOVE_NETWORK`: add a network configuration block, equivalent to the file configuration blocks.
- `SELECT_NETWORK`: attempt authentication and association to a configured network.
- `MIB`: get a listing of IEEE 802.11 (`dot11*`) and 802.1X (`dot1x*`) Management Information Base (MIB) attributes regarding various authentication state parameters.
- `STATUS` and `STATUS-VERBOSE`: request current WPA/EAPoL/EAP status information.

The same control socket is used for unsolicited event messages from the supplicant. Events may be received on the socket between command requests/responses. To avoid request/response pair breaking, the documentation suggests either one of two options: using two different control socket connections, one for commands/responses and another for unsolicited events; or multiplexing both in the same connection, supplying a different callback handler for each request command. Unsolicited event messages obviate the need to poll variables for occurrences such as EAP success or failure events. Some events are mere translations of kernel networking events, which render *wpa_supplicant* a viable option as a slightly higher level interface for wireless device control.

The D-Bus interface for *wpa_supplicant* was developed as a Google Summer of Code project in 2009[1]. As usual with D-Bus, it offers an object-oriented view of the *wpa_supplicant* software. There are basically five interfaces to the wpa_supplicant D-Bus API:

- `fi.w1.wpa_supplicant1`: offers methods to add, remove or list network devices to further control.
- `fi.w1.wpa_supplicant1.Interface`: represents a network device. Network configuration blocks are added or handled through this interface. This also allows requesting an interface to connect, disconnect or scan for available networks. As a result, this is also where most relevant signals are generated, like BSS detection following scans.
- `fi.w1.wpa_supplicant1.Interface.WPS`: used to handle WPS procedures.

[1]Google Summer of Code *wpa_supplicant* proposal, `http://google-melange.appspot.com/gsoc/project/google/gsoc2009/wsowa/2001`

- `fi.w1.wpa_supplicant1.BSS`: interface for scanned BSS objects. Attributes of an AP such as the frequency, signal strength and supported authentication and encryption methods are available in this interface.

- `fi.w1.wpa_supplicant1.Network`: implements a network configuration block. This is just a key-value map of strings containing the same information as the network blocks in the configuration file.

wpa_ supplicant is a sub-project of *HostAP*[1], which is basically a driver plus a user space daemon (*hostapd*) for access point hardware. The daemon controls Station authentication and supports the same methods as *wpa_ supplicant*. Both are inter-operable, but they are independently tested as well. This is currently the most supported authentication supplicant for GNU/Linux based devices, including the Android OS.

3.3.4 DNS

In GNU/Linux, there is no specific tool to handle DNS servers. The list of known servers is configured directly in a file usually located at `/etc/resolv.conf`, which the *resolver* library parses in order to know where to send DNS queries. There is a framework, *openresolv*[2], to manage the `resolv.conf` file from multiple sources such as DHCP clients. It provides a program that can be invoked through the command line and add DNS configurations for each interface, which will then update the `resolv.conf` file as it thinks best. This tool is mostly unused because it requires a greater effort than simply writing the configurations directly to the file. Also, since most network configuration setups are either manual, or relying on tools that handle all the interfaces, it is easy to keep track of multiple DNS configurations for each network without additional tools.

3.3.5 DHCP clients

When it comes to automatic network layer configuration, GNU/Linux machines mostly rely on two options: *dhclient*[3] and *dhcpcd*[4]. The first one is part of a client-server software bundle provided by the ISC. The project's website claims it "the most widely used open source DHCP implementation on the Internet". The bundle is regarded as a reference implementation of the protocol, and quoted as a "production-grade software, suitable for use in high-volume and high-reliability applications". It is actively maintained, and the ISC provides paid support for the software. *dhcpcd* is a lightweight alternative that is supported by most, if not all GNU/Linux distributions. It uses the Netlink kernel interface for configuration, while the *dhclient* software remains dependant on ioctl calls.

[1]HostAP, `http://hostap.epitest.fi/`

[2]openresolv, `http://roy.marples.name/projects/openresolv`

[3]*dhclient*, `https://www.isc.org/software/dhcp`

[4]*dhcpcd*, `http://roy.marples.name/projects/dhcpcd`

The major difference between both clients refer to the supported standards. *dhcpcd* supports IPv6 stateless autoconfiguration, but it does not implement the DHCPv6 protocol, while *dhclient* does. Command line usage of both applications is similar. By default, both work as daemons and automatically take care of lease acquisition and renewal. Configuration can be based on text-files or through command line options.

There are differences regarding programming interfaces as well; the *dhclient* manual pages mention controlling the daemon through their OMAPI framework, but the framework documentation refers only to communication with the DHCP server. The *dhcpcd* daemon, however, offers two interfaces: a UNIX domain socket and a D-Bus interface. The socket interface is undocumented, and the D-Bus interface is a merely a proxy for it. The D-Bus interface receives events from the socket and propagates them as D-Bus signals to its listeners, and also supports several operations: retrieving lease information and current status, renew and release leases, and more.

There are other DHCP implementations for GNU/Linux, but not as common, and lacking advanced features.

3.4 Full-featured network management Solutions

By combining the tools described in the previous section it is possible to build a complete solution that handles all the steps for network configuration. Some GNU/Linux distributions like Debian[1] or Fedora[2] offer a set scripts to automatically attempt to configure a network once an interface is powered up. This is easy to maintain for stationary hosts, but laptop users, that usually attach to several different networks throughout the day or week, require more interactivity and ease of use. This is often achieved with graphical user interfaces, and integration with the DE through applets. Two of the most popular solutions for network management are presented in the following sections.

3.4.1 wicd

wicd[3] aims to provide a simple interface for a wide variety of network configuration settings. Its first version was developed in 2006 under the name "Connection Manager" using the Python programming language. It is split into two major parts: a daemon and a user interface. The daemon is responsible for handling the configuration and log files, as well as effectively managing the connections. The user interface is composed by a tray icon and a GUI window (Figure 3.6). The graphical configuration interface is built with the GIMP Toolkit (GTK+) and there are no Desktop Environment dependencies, which makes it usable wherever GTK+ is available. There are also two console interfaces: a regular command line version, and a curses variant that displays a visual interface on character cell terminals. All of these interfaces connect to the daemon through a D-Bus interface, which is not documented.

[1]Debian, `http://www.debian.org/`

[2]Fedora, `http://fedoraproject.org/`

[3]wicd, `http://wicd.sourceforge.net/`

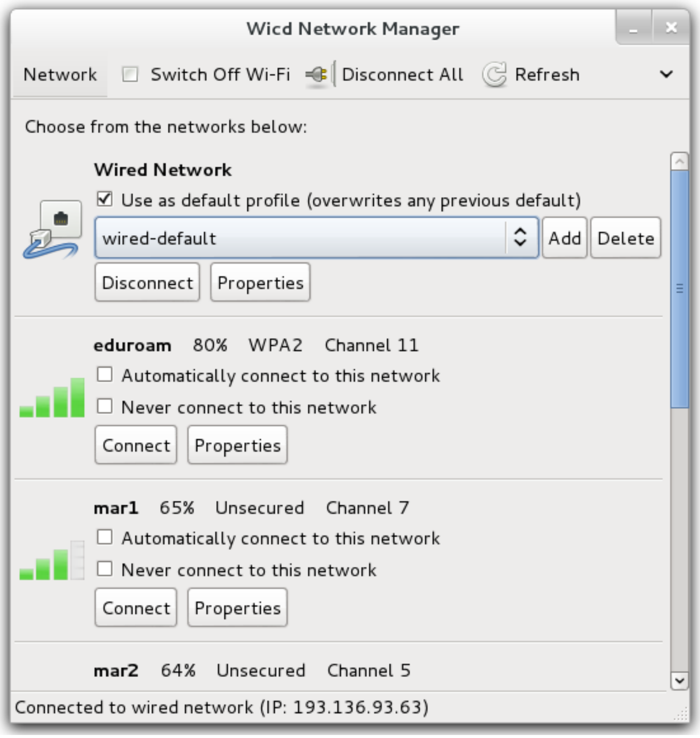

Wicd Network Manager
Network
Switch Off Wi-Fi
Disconnect All
Refresh
Choose from the networks below:
Wired Network
Use as default profile (overwrites any previous default)
wired-default
Add
Delete
Disconnect
Properties
eduroam 80% WPA2 Channel 11
Automatically connect to this network
Never connect to this network
Connect
Properties
mar1 65% Unsecured Channel 7
Automatically connect to this network
Never connect to this network
Connect
Properties
mar2 64% Unsecured Channel 5
Connected to wired network (IP: 193.136.93.63)

Figure 3.6: *wicd*'s graphical user interface window.

The daemon does not talk directly to the kernel to handle network configuration. Instead, it depends on other tools for network management, mostly in a scripted manner. The dependencies include the following list: *ethtool* or *mii-tool*, *net-tools*, *dhcpcd* or *dhclient*, *inetutils* or *iputils*, *wpa_supplicant* and *wireless_tools*. *ethtool* and *mii-tool* have not been mentioned before. They offer commands for modifying various low level network device parameters such as speed, duplex, flow control, and even perform firmware upgrades on the devices. *wicd* uses them simply to check if there are cables attached on Ethernet interfaces. *inetutils* and *iputils* have also not been mentioned before. They are a collection of network programs that provide the *ping* utility. *wicd* tries to ping the default gateway after completing a connection, to determine the success of the operation. *wpa_supplicant* is used for authentication, while *net-tools* and either DHCP client referred in section 3.3.5 are used for static and dynamic IP configuration, respectively.

wicd has many users, but it is built upon a collection of tools that were long deemed unfit to the current GNU/Linux infrastructure. This should not presently affect the end user, but is a shortcoming for future development and support.

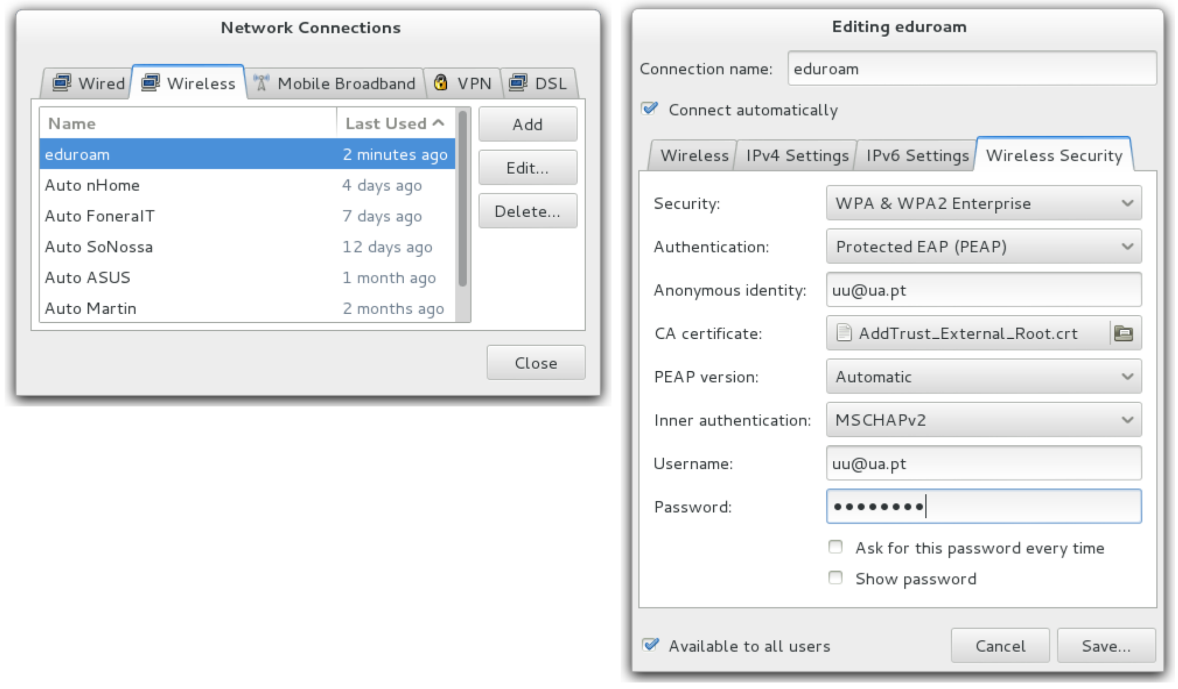
3.4.2 NetworkManager

NetworkManager[1] is an even more complete solution for GNU/Linux network managing. Compared to *wicd*, it supports the same authentication and IP configuration features, but it also adds extra features like Virtual Private Network (VPN) configuration support. It of-

[1] *NetworkManager*, `http://projects.gnome.org/NetworkManager/`

fers an architecture of separate daemon and user front-ends for configuration as well. There are various available front-ends including graphical interfaces (Figure 3.7), tray applets (Figure 3.8) and command line interfaces. The daemon offers a D-Bus self-documented programming interface that is used by the various front-ends. This view of the architecture shows a great similarity between *NetworkManager* and *wicd*, but there are very important differences. While *wicd* supports only Ethernet and Wi-Fi devices, *NetworkManager* also supports WIMAX, Modems (GPRS, UMTS, etc) and Bluetooth. Moreover, it avoids depending on command line tools when feasible.

The core of *NetworkManager* implements direct control of network interfaces through the kernel mechanisms, and follows a very structured approach regarding those interfaces. It is developed in the C language, but follows an object-oriented solution with the help of the GLib libraries. The D-Bus interface is managed by the GLib bindings, so the internal architecture of the data structures is directly reflected to the D-Bus interface. The most relevant structures of the API are described in the following sections.



Name	Last Used
eduroam	2 minutes ago
Auto nHome	4 days ago
Auto FoneraIT	7 days ago
Auto SoNossa	12 days ago
Auto ASUS	1 month ago
Auto Martin	2 months ago

Connection name: eduroam

[✓] Connect automatically

Security:	WPA & WPA2 Enterprise
Authentication:	Protected EAP (PEAP)
Anonymous identity:	uu@ua.pt
CA certificate:	AddTrust_External_Root.crt
PEAP version:	Automatic
Inner authentication:	MSCHAPv2
Username:	uu@ua.pt
Password:	••••••••

☐ Ask for this password every time

☐ Show password

[✓] Available to all users

Figure 3.7: *NetworkManager*'s configuration GUI.

3.4.2.1 org.freedesktop.NetworkManager

A main central object offers an interface for overall management tasks. This object holds the network device structure, the overall machine state and connection states. The following mechanisms are available:

- **Add** or **Get** a list of devices for network management.
- **Add**, **Activate** or **Deactivate** network connections.

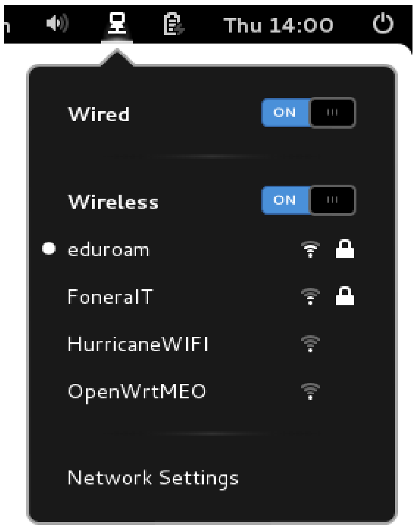


Figure 3.8: Gnome's *NetworkManager* applet.

- **Enable** or **Disable** networking, with the ability to specify individual technologies (i.e., only Wi-Fi, WIMAX or Modem).
- Check the overall **network state**.

The network state can vary between one of the following: unknown, asleep, disconnected, disconnecting, connecting, connected_local, connected_site and connected_global. Although the interface distinguishes the connectivity scope, the current version of *NetworkManager* actually always assumes that, when there is connectivity, it is global. The main object will emit signal for any property or state changes that occur, and any time a device is added or removed.

Internally, this object is mainly responsible for maintaining the global state machine that controls the connection procedures and the remaining data structures.

3.4.2.2 org.freedesktop.NetworkManager.Device

This is an abstract interface to represent common attributes of a network device. An object implementing this interface is related to a unique network device in the system. Every network device has a device type, a current state, a list of capabilities, etc. The device type is self-explanatory, and identifies which of the supported device types this object refers to. The following list of states is defined by the Device interface: unknown, unmanaged, unavailable, disconnected, prepare, config, need_auth, ip_config, ip_check, secondaries, activated, deactivating and failed. The interface and defines signals to indicate Device changes state.

Despite the object-oriented architecture of the program, a user does not "ask a device to connect to a certain network". Instead, a user requests the *NetworkManager* to "connect this device to a specific network". The difference is that, instead of requesting a connection through the device object, a request is made to the central object indicating both the connection details and the device to be used. However, requests to Disconnect are made directly to the Device interface.

Most operations regarding the physical device manipulation are handled through the kernel's Route Netlink interface. Bringing interfaces up and down, configuring IP addresses, routes, etc, is common code to all device types.

3.4.2.3 org.freedesktop.NetworkManager.Device.Wired

The specific Wired interface is very basic and, on top of the abstract Device interface, it only adds attributes such as the MAC address, the design data rate of the device and whether a cable is plugged in or not. This data structure supports the distinction between a permanent hardware address and a software hardware address that may be configured on the device. Both will always match if the driver does not support address changes. Any property change will be signalled (including changes in properties inherited from the Device interface).

Getting or setting a network interface's data rate is not available through the Netlink protocol. *NetworkManager* depends on *ethtool* or *mii-tool* for this task. They are also used for carrier detection whenever a specific driver does not implement the flag through the Route Netlink protocol.

3.4.2.4 org.freedesktop.NetworkManager.Device.Wireless

Other than the MAC address attributes, similar to the Wired interface, this interface has specific methods to handle AccessPoint objects. It is possible to request a wireless device to scan for APs through the D-Bus interface and fetch the known AccessPoint list. Any AccessPoint removal or addition is announced by a signal, alongside any other Device property changes. Other specific attributes include the current device bitrate, the Active AccessPoint, and wireless capabilities of the device. The capabilities field indicates the supported encryption mechanisms such as WEP, TKIP, CCMP, WPA and WPA2.

NetworkManager uses three different mechanisms for kernel interfacing: WEs, nl80211, and *wpa_supplicant*. WEs is only used as a fallback if nl80211 is not supported. The nl80211/WEs duo is used for getting current link attributes such as SSID, frequency, data rate, link quality, etc. In this context, the wpa_supplicant API is used to trigger scans and handle scan results.

3.4.2.5 org.freedesktop.NetworkManager.AccessPoint

The AccessPoint object originates from WirelessDevice objects. An AccessPoint object exposes properties to identify networks and determine how to properly prepare the association to the Basic Service Set, according to the following list:

- **SSID**: the Service Set Identifier of the Access Point.
- **Frequency**: the operating frequency of the AP.
- **Hardware Address:** the MAC address of the AP.

- **Strength**: the current signal quality of the AP, in percentage. *NetworkManager* calculates this from the hardware-reported *dBm* value, blindly normalizing it to a scale between $-40dBm$ (best, 100%) and $-100dBm$ (worst, 0%), and linearly adjusting the intermediate values to a percentage.
- **Maximum Bitrate**: the maximum supported bitrate of the AP.
- **Mode**: this is usually "infrastructure", but could be "ad-hoc" if the object actually represents an IBSS node.
- **Security** and **Privacy** flags: these indicate whether the AP supports security and privacy mechanisms and, if so, which.

Property changes are also signalled to D-Bus clients; the only dynamic attribute for the AccessPoint objects should be signal Strength.

3.4.2.6 org.freedesktop.NetworkManager.Settings

The Settings interface does not handle global *NetworkManager* settings *per se*. The Settings interface actually deals with network connection settings. There is only one object implementing this interface, and it may be regarded as the system-global repository for the different configured networks. As such, it exposes methods to add and retrieve connection objects, and signalling new additions to every client. This interface does not offer methods to remove connection objects from the repository; instead, the method is available in each connection object itself.

The Settings manager stores the network configurations in plaintext files, one for each connection. These files are only accessible by the system administrator. The passwords are stored along with the other settings, but *NetworkManager* also supports letting the user store them on a keyring, as long as it provides a pre-defined D-Bus interface for retrieving the secrets. Alternatively, secrets can be requested to the user whenever they are required.

3.4.2.7 org.freedesktop.NetworkManager.Settings.Connection

This interface is implemented by objects that contain the necessary information on how to connect to a given network. The complete specification of the possible settings is a very extensive list of key-value pairs divided in groups. Some of those groups follow:

- `connection`: this setting is required. Here the key-value pairs help identify the connection in the overall listing, by assigning a name and unique ID to the configuration.
- `802-3-ethernet`: this setting holds specific ethernet configurations for the network, including the desired data rate, software MAC address, and MTU size.
- `802-11-wireless`: for IEEE 802.11 wireless networks, the network SSID is specified here. It also allows indicating specific MTU size, cloned MAC address, data rate, and additionally restricting association to a specific AP by providing its MAC address.

- `802-11-wireless-security`: when an IEEE 802.11 network employs security mechanisms, this setting stores the configuration for those mechanisms.
- `802-1x`: if the security mechanisms of a connection require the IEEE 802.1X standard, the authentication elements such as identities, passwords, keys and certificates are configured here.
- `ipv4`: IP configuration can be dynamic, by choosing DHCP, or static, in which case it is possible to indicate a list of desired IP addresses, DNS servers, and static routes.
- `ipv6`: this setting is similar to `ipv4`, except the parameters are parsed differently because of the different address representations.

Dynamic IP configuration, is not handled by *NetworkManager* directly; it relies on either *dhcpcd* or *dhclient*, whichever is available. DHCPv6 will not be supported if only *dhcpcd* is available. The authentication and security configurations are translated to the *wpa_ supplicant* network block format of key-value pairs, which is mostly equivalent to *NetworkManager*'s own format.

3.4.3 Acceptance

NetworkManager is a project maintained by Gnome. However, given its maturity and stability, it is the default network management solution for many GNU/Linux distributions. While many Desktop Environment attempt to provide their own software implementations for text editors, console terminals, etc, *NetworkManager* is a common choice among many other projects including KDE, Xfce[1], and Lightweight X11 Desktop Environment (LXDE)[2].

3.5 Conclusion

This chapter covered most of the frameworks and tools that are used for network management. This analysis focused on the tools and frameworks that are currently recommended for software development in GNU/Linux. In summary, networking tools should use Route Netlink for interface state and IP layer management, and nl80211 for Wi-Fi specific operations. DE integration requires usage of the D-Bus IPC framework. The *NetworkManager*'s D-Bus API is sufficiently generic for any Network Manager to implement, and would enable reuse of many applets and programs that use the API.

[1]Xfce, `http://www.xfce.org/`
[2]LXDE, `http://www.lxde.org/`

Chapter 4

IEEE 802.21

The IEEE 802.21 group stemmed from a discussion in the IEEE 802.16 group regarding network handoffs, aiming at a common layer 2 handover interface between IEEE 802 as well as non-802 systems, thus a Media Independent Handover (MIH) framework. This chapter gives an overview of the standard, and a major open-source implementation, Open Dot Twenty ONE (ODTONE)[1]. The chapter concludes with proposed extensions in order to effectively support network management operations.

4.1 Motivation

Handovers between points of access of the same technology are largely discussed in each standard. These are called horizontal handovers, because the link layer does not change. The handover decision is usually handled by the Mobile Node (MN), in regard to timing and target selection. Access Point selection in IEEE 802.11 networks, for example, is usually based on received signal strength and/or interference parameters. This is true for both when initially joining the network as well as for link loss or degradation events. More often than not, the AP with the seemingly best signal qualities is serving a greater number of users, thus causing congestion and ending up providing the least appropriate service when compared to an hypothetical AP with weaker signal, but less users. Since the network takes no part in preparing the handovers, it is common that existing connections face a great number of lost packets before the terminal establishes the new link and requests new connections.

The problem of network handoffs increases when the transition happens between different technologies. In order to accommodate the growing number of users and greater bandwidth requirements for wireless Internet access, cell radius tends to decrease, while the coexistence of heterogeneous networks enables better service provision, for example by combining the higher bandwidth and lower cost of Wi-Fi with the better mobility and larger coverage of cellular networks [53]. In these cases, handovers are not limited to a link layer change in the point of access, but possibly the whole network infrastructure, depending on the technology and the service providers. Such is usually the case of transitioning from home or office

[1]ODTONE, `http://atnog.av.it.pt/projects/odtone`

IEEE 802.11 networks to cellular networks. These are called vertical handovers, because the transition occurs between different link layers, and connectivity must be reestablished at the network and/or transport layers. Again, it is usually the MN that decides the timing and target network, based on user configuration or perceived signal quality metrics. 3GPP supports network initiated handover processes, but does not include handing over to different technologies.

The IEEE 802.21 protocol defines a common sublayer directly on top of the link layer, for adoption among 802 and non-802 network technologies, for both horizontal and vertical network handovers. This protocol aims to provide handover operation at layer 2 between any number of technologies, while addressing factors that condition handover decisions such as service continuity, QoS, application specific tolerances, network discovery and selection, and power management. Furthermore, the technology defines procedures for MN-initiated, MN-controlled, Network-initiated and Network-controlled handovers.

4.2 Architecture

The IEEE 802.21 standard enables cooperation on both the mobile node and the network by providing a framework for information exchanges between both parts. There are several network entities taking part in this framework, represented by Figure 4.1. Apart from the Mobile Node itself, the following entities are identified in the network:

- **MIH Point of Attachment (PoA)**: the endpoint to which the MN attaches when joining the network.
- **MIH Point of Service (PoS)**: a network entity that exchanges MIH messages with the Mobile Node. It helps the MN perform handovers by providing target network information as well as preparing networks for attachment.
- **MIH non-PoS**: a network entity (usually database servers) that does not exchange messages with the Mobile Node. Usually contain geographical and administrative network information.

Every MIH capable device implements a Media Independent Handover Function (MIHF); it is the main point of interaction between the upper and lower layers, and coordinates the exchange of commands and information between the different MIH entities, as indicated by Figure 4.2. Communication between entities and the MIHF is defined by Service Access Points (SAPs). Entities in higher layers are called MIH Users, and communicate with the MIHF using the MIH_SAP interface. Lower layer entities are called MIH Links and communicate with the MIHF via the MIHF_LINK_SAP interface. MIH Users and Links can only communicate with the MIHF local to the device where they are present; remote communication is achieved by interconnecting multiple MIHFs, via the MIH_NET_SAP interface.

The MIHF entities in MN and Network entities communicate with each other using specific messages, composing the MIH protocol. These messages correspond to primitives of the MIH

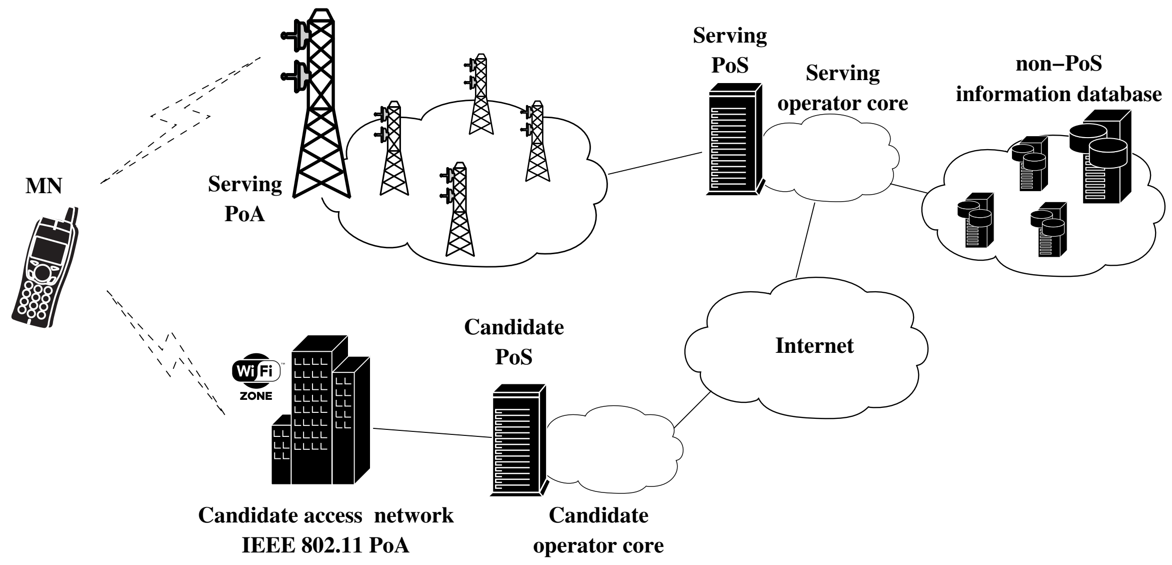


Figure 4.1: IEEE 802.21 Reference Model

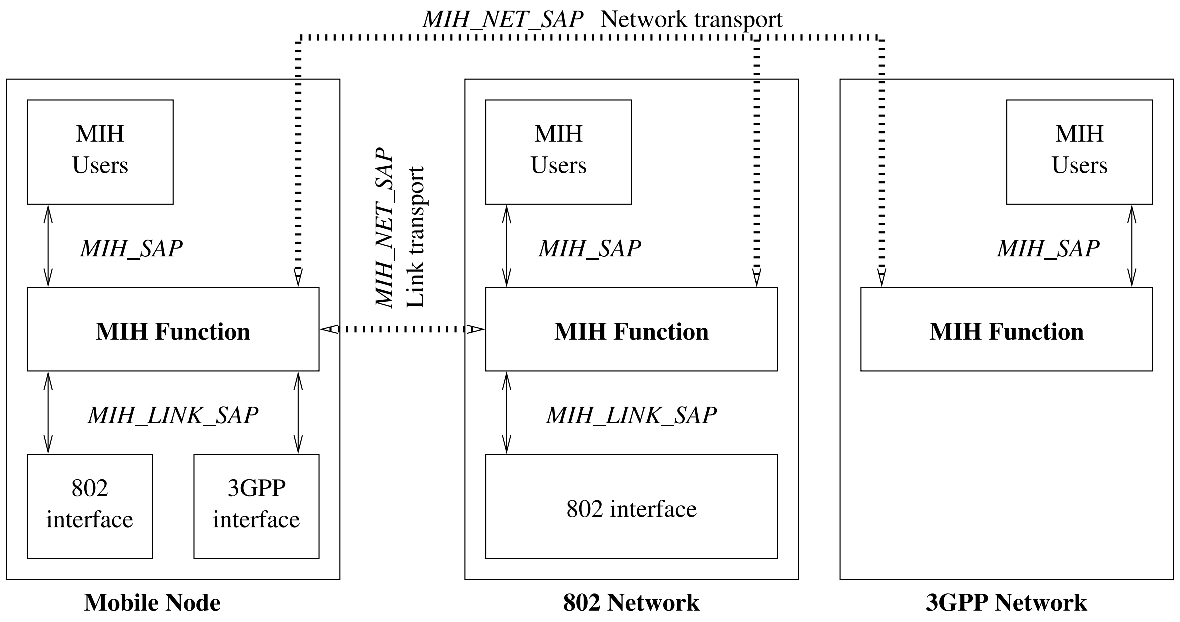


Figure 4.2: IEEE 802.21 General Architecture.

Services, and always contain two identifiers: an MIHF ID and a Transaction ID. The payload for messages is encoded in the TLV format defined in the standard.

The standard does not define policies and algorithms for handover decisions. MIH Users are responsible for this task, by making use of the MIHF in order to communicate with the available entities and obtain relevant information on which to act. MIH communications available on each SAP fall into three different service categories, explained in the following sections.

4.2.1 Media Independent Event Service

In general, handovers can be initiated either by the MN or by the network. Events relevant to handover originate from Links at the MN, or from the entities in the network. Hence, the source of these events is either a local or remote entity. Multiple higher layer entities can be interested in these events at the same time. The MIHF tracks event subscriptions from higher layers and dispatches the events to the correct destinations. Events generated by the MN Links are called Link Events, and delivered to the MIHF through the MIH_LINK_SAP. Once received at the MIHF, they are translated into MIH Events for delivery to MIH Users through the MIH_SAP. The following Link events are available:

- **Link_Detected**: indicates the presence of a new PoA, which implies the MN entered its coverage area. This event is not generated for additional PoAs of the same network.
- **Link_Up**: delivered when a layer 2 connection is established on the specified link interface.
- **Link_Down**: generated when a layer 2 connection is no longer available for sending frames. Roaming between PoAs of the same network does not generate this event.
- **Link_Parameters_Report**: indicates changes in link conditions that have crossed specified threshold levels. It may also be generated at specified intervals for various parameters.
- **Link_Going_Down**: issued when a Layer 2 connection is predicted to go down (Link_Down) within a certain time interval.
- **Link_Handover_Imminent**: generated when a native link layer handover or switch decision has been made and its execution is imminent.
- **Link_Handover_Complete**: indicates that a native link layer handover/switch has just been completed.
- **Link_PDU_Transmission_Status**: indicates the transmission status of a higher layer Protocol Data Unit (PDU) by the link layer.

4.2.2 Media Independent Command Service

The Media Independent Command Service (MICS) refers to the commands sent from the higher layers to the lower layers in order to control or determine the status of links, or to initiate handover procedures. Commands on the MIH_SAP interface are called MIH Commands; commands on the MIH_LINK_SAP are called Link Commands. Link Commands are just a translation of MIH Commands that the MIHF sends to the lower layers on behalf of the MIH Users. Link commands are local only, MIH commands may be issued remotely for handover initiation. A brief overview of the various Link Commands follows:

- **Link_Capability_Discover**: query the list of supported link layer events and commands.
- **Link_Event_Subscribe**: subscribe one or more events from a specific link layer technology.
- **Link_Event_Unsubscribe**: unsubscribe from a set of previously subscribed link-layer events.
- **Link_Get_Parameters**: obtain the current value of a set of link parameters from a specific link.
- **Link_Configure_Thresholds**: configure thresholds and/or specify the time interval between periodic reports for the Link_Parameters_Report event.
- **Link_Action**: order the link layer to shut down, to remain active, to perform a scan, or to come up active and remain in stand-by mode. An execution delay time can also be specified.

MIH Commands include the specified Link commands, with the addition of the following remote commands, where the "MN" prefix refers to commands from the MN to the Network; the "Net" prefix for Network to MN commands; and "N2N" Network to Network commands:

- **MIH_Net_HO_Candidate_Query**: initiate handover and send a list of suggested networks and associated points of attachment.
- **MIH_MN_HO_Candidate_Query**: query and obtain handover related information about possible candidate networks.
- **MIH_N2N_HO_Query_Resources**: used by an MIHF on the serving network to communicate with its peer MIHF on the candidate network.
- **MIH_MN_HO_Commit**: notify the serving network of the decided target network information.
- **MIH_Net_HO_Commit**: request the MIH user to perform a Network-controlled or Network-assisted handover based on selected choices for candidate networks and specific PoAs.
- **MIH_N2N_HO_Commit**: inform a selected target network that an MN is about to move to the target network.
- **MIH_MN_HO_Complete**: indicate the completion of MIH level handover aiding procedure.
- **MIH_N2N_HO_Complete**: used by an MIH user in the network to communicate with a peer network MIH entity about the completion of handover operation.

4.2.3 Media Independent Information Service

The Media Independent Information Service (MIIS) provides a framework by which an MIHF, residing in the MN or in the network, discovers and obtains network information within a geographical area to facilitate network selection and handovers. MIIS is based on IEs which provide information that is essential for a network selector to perform “intelligent” handover decisions. This information can be related to neighbor maps, coverage zones, availability of Internet connectivity or specific services, etc. This information can be used for the discovery of available networks in a geographic area, for example, without the need to power additional interfaces and perform scans. Dynamic information should be obtained through the previously described Event and Command services.

There are basically two actions associated with the MIIS:

- **MIH_Get_Information**: request information from an MIH information server.
- **MIH_Push_Information**: used by an MIIS Server to push information to the MN.

4.2.4 Media Specific Mappings for SAPs

The MIHF aggregates disparate interfaces with respective media dependent lower layer instances into a single interface with the MIH Users, reducing the inter-media differences to the extent possible. For the most part, existing primitives and functionality provided by different access technology standards are used. Amendments to existing standards are proposed where necessary to fulfill the MIHF capabilities.

The Link Service Access Point (LSAP), defined in the IEEE 802.2 standard, provides the interface between the MIHF and the Logical Link Control (LLC) sublayer across both IEEE 802.3 and 802.11 networks. The MIH_LINK_SAP set of primitives requires additional support that maps to 802.11 through an enhancement proposal defined by IEEE 802.11u [54]. This means that most currently available 802.11 hardware does not yet implement the necessary primitives for full MIH compatibility.

4.3 Open Dot Twenty ONE

Open Dot Twenty ONE, or ODTONE, is an open source implementation of the IEEE 802.21 Media Independent Handover framework. ODTONE supplies an implementation of the MIHF, supporting the whole range of MIH services. It also provides a set of APIs, enabling the development of custom MIH Users and Links to interface with the provided MIHF. ODTONE is implemented in C++, aided by the boost libraries[1], but the modules are decoupled and communicate with each other using network sockets.

Data transport between remote MIH entities is based on the TLVs defined in the standard, but communication for local entities is not defined, since the standard allows monolithic implementations. For this, ODTONE reuses the existing TLVs where possible, defining new

[1]Boost libraries, `http://www.boost.org/`

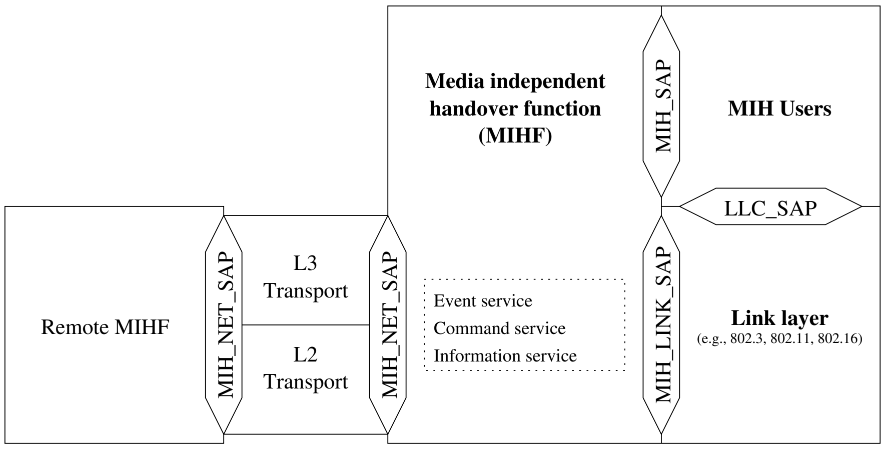


Figure 4.3: General MIHF reference.

ones where necessary. This decoupled nature means that it is possible to deploy the same MIHF implementation in multiple Operating Systems, while only the Link SAPs would depend on different OS mechanisms. The same is true for MIH Users, in the case that they interact with other desktop applications. The core of the ODTONE framework is compiled and tested in Microsoft Windows and GNU/Linux Operating System, including the Linux-based Android OS.

4.4 Extending 802.21 towards Media Independent Network Management

The IEEE 802.21 provides a link layer abstraction for control and information gathering from various access interface technologies. This enables the implementation of a network management application agnostic to interface technologies. However, the general MIHF Reference Model and SAPs definition, illustrated in Figure 4.3, does not supply a complete abstraction between the higher and lower layers. In fact, the MIH framework does not offer primitives for MIH Users to initiate authentication or association to a certain network, for example. Leaving such procedures out of scope requires the ability for higher layers to bypass the MIH layer even for handover-specific procedures. Moreover, there is no support for network layer configuration.

The MIHF qualifies as higher layer component, since it uses transport layer mechanisms for communication. In fact, ODTONE is a userspace implementation of the framework, and can be extended to provide an abstraction for the Operating System's network configuration subsystem. Figure 4.4 presents a perspective of the extended 802.21 standard as a framework for network management using ODTONE.

Since ODTONE provides only an MIHF implementation, the remaining Link SAPs and MIH Users have to be implemented as well. The standard is unclear as to where the components are positioned in relation to the OS kernel. In fact, the positioning of mechanisms such as IP addressing and authentication is not important, although in terms of performance it would be ideal that the Link SAPs were as close as possible to the hardware and OS mechanisms. Given the nature of the IPC mechanisms for the ODTONE framework, and looking to provide greater flexibility and support, the whole architecture can be implemented at userspace, interfacing with the kernel mechanisms where necessary.

The additional MIH primitives for network management operations are described next.

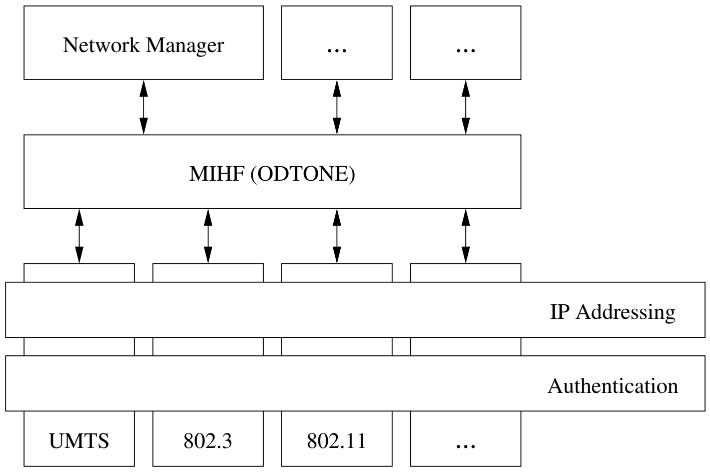


Figure 4.4: IEEE 802.21 architecture for network management.

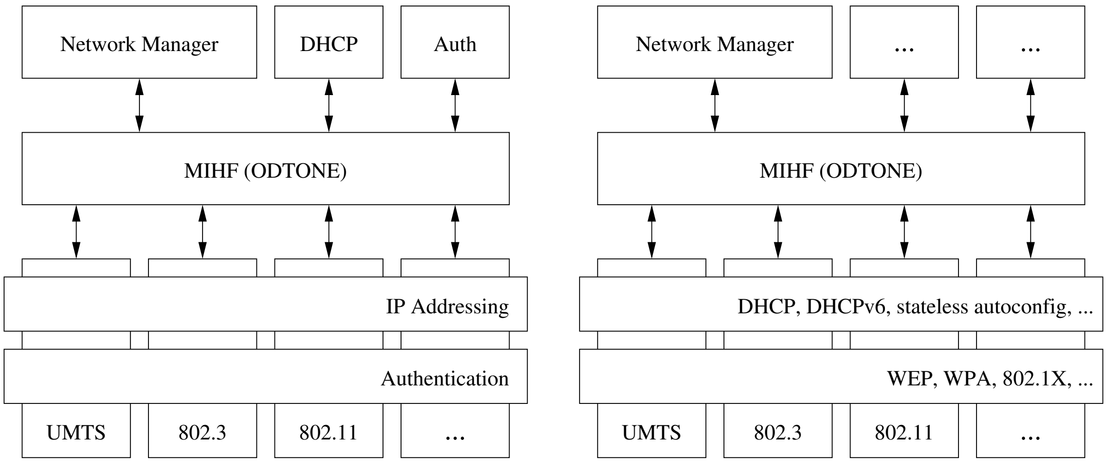
4.4.1 IEEE 802.21 extensions

The proposed extension refers to operation within the Mobile Node only. Association and security procedures require link layer communication from supplicant software. IP configuration requires adding IP addresses to interfaces, as well as default gateways, custom routes, and DNS servers. IEEE 802.21 does not provide such a direct control. As such, two additional Command Service primitives are proposed for these tasks, allowing for Network Management application scenarios and providing the following functionality:

- **Link_Conf**: attach to a given network, providing the necessary authentication, association and security information.
- **L3_Conf**: configure a set of networking properties on an interface, such as IP address, static routes and list of DNS servers.

While it is possible to use these commands for the referred tasks, handling network security mechanisms is not so straightforward. Network authentication protocols such as EAP provide many authentication mechanisms, some requiring an indefinite number of exchanges between the supplicant and the authentication server, and possibly demanding user input. This can be interpreted as a network request for the user, and can be implemented through the 802.21 Event Service. The following primitive is proposed:

- **Link_Conf**: indicates a network request for authentication material from the supplicant.



(a) MIH User centered. (b) Link SAP centered.

Figure 4.5: Architectures for 802.21 based Network Manager.

Once again, there is no indication as to where certain software components should be placed, this time in relation to the MIH architecture. For example, the authentication supplicant could be an MIH User using Link_Conf commands to trigger communications with the EAP authentication server; or it could be tightly coupled with the Link SAP and receive the required authentication parameters through the Link_Conf command. The same question affects components such as DHCP clients. A DHCP client could implement an MIH User and provide the Link SAP with just the resulting information, or the Link SAP itself could implement these protocols and perform the operations. Figure 4.5 clarifies the two scenarios, where 4.5a shows the option where both authentication supplicant and DHCP applications are MIH Users, and 4.5b shows these components at the Link SAPs level.

Interpreting Figure 4.5a leads to conclusion that having DHCP clients or authentication supplicants as MIH Users requires synchronization between the Users themselves. For example, a Network Manager initiates a connection, and it must tell the authentication supplicant to handle authentication. Also the DHCP client could run after the Link SAP informs of layer 2 connectivity completion, but not all networks provide/require DHCP configuration. This must be triggered by the network management software. This type of synchronism requires additional MIH service extensions, out of band communication, or a monolithic Network Manager with DHCP and security support.

Figure 4.5b, on the other hand, would only require that the Link_Conf and L3_Conf messages indicate whatever actions need to be carried out, and let each Link handle the tasks. Overall, a Link centered solution would require less effort or modification to the MIH standard. Support for each protocol need not be implemented per-Link, it is still possible to have separate components that the Link SAP can use when necessary. This scenario favors the fact that existing applications are already tailored for use as a service. For example, it is easy to grab an authentication supplicant software and use it for authentication, but it is much harder to change that software to use the MIH infrastructure.

As such, the message format for each primitive is proposed as follows:

- **Link_Conf**: due to the high number of available authentication mechanisms, it is counterproductive to define data structures for each and every possible combination. As such, the Link_Conf message structure consists of the following:
  - A **LINK_TUPLE_ID**: this is only for the MIH_SAP interface, and allows the MIHF and MIH Users to identify the Link the message is intended to.
  - A **LINK_ADDR**: this data type is defined by the 802.21 standard, and helps identify a network via the link layer address of its PoA.
  - A list of **CONFIGURATION** elements: the CONFIGURATION data type is proposed in light of this message, and consists of a pair of two OCTET_STRING elements. The first element is intended as a "key", whereas the second element holds a "value" for that key. This type of structure is commonly referred to as a map, dictionary, or associative array. This mechanisms allows passing all kinds of information, whether textual, numeric, or binary, as long as the MIH Users and Links agree on the interpretation of each key.
- **L3_Conf**: this message format is easier to define with stricter data types:
  - **LINK_TUPLE_ID**: similarly to the Link_Conf message, this parameter is only included for messages exchanged via the MIH_SAP interface.
  - **IP_CFG_MTHDS**: this data type is defined in the 802.21 standard. It consists of a bitmap that allows choosing one or more network layer configuration methods, including static, stateless and dynamic configuration, as well as using mobility mechanisms. Dynamic configuration is interpreted as DHCP, and it is possible to indicate the desired version separately.
  - Optional **IP_MOB_MGMT**: if mobility mechanisms are requested, this parameter identifies the specific method that is desired. This is also defined by the 802.21 standard.
  - Optional list of **IP_INFO**, for device addresses: the IP_INFO data type is proposed for these new messages, and is composed by three elements:
    - **IP_ADDR**: the IP desired address for the interface.
    - **IP_PREFIX_LEN**: the network prefix length for the subnet.
    - **IP_ADDR**: the default gateway for the subnet.
  - Optional list of **IP_INFO**, for routes: in this context, the IP_INFO elements take the following meaning:
    - **IP_ADDR**: the address of the target network.
    - **IP_PREFIX_LEN**: the prefix length for the address of the target network.
    - **IP_ADDR**: the gateway for that network.

  - Optional list of **IP_ADDR**, for DNS servers.
  - Optional list of **FQDN**: an FQDN is basically an OCTET_STRING that represents a domain name. This field indicates the domain names to which the terminal belongs.

- **Link_Conf_Required**: this message presents similarities with the Link_Conf primitive, since it refers to the same context. The following elements are present:
  - **LINK_TUPLE_ID**: used to identify the link in question.
  - List of **CONFIGURATION**: the same concept of "key-value" pairs is applied to the network requests for authentication information.

The proposed parameters for each message support all of today's usual connection management settings for computers and other types of internet clients.

4.5 Conclusion

With the proposed extensions, the MIH infrastructure can be effectively used by network management applications to fully control interfaces and make network selection decisions based on multiple parameters provided by the 802.21 framework. This includes the usual terminal-based selection mechanisms that rely on network conditions such as the signal to noise ratio, but also the network information that allows selection based on user and application requirements ranging from cost, latency and bandwidth limits, supported services, etc.

The following chapter presents an implementation of a framework that integrates the Media Independent mechanisms provided by 802.21 with the current desktop design for Network Manager applications, which is a first step towards an Enhanced Media Independent COnnection Manager: EMICOM.

Chapter 5

EMICOM Implementation

The tools and technologies for integration in this network management architecture were introduced in the previous chapters. This chapter describes the implementation of such an architecture, based on those tools. The aim of this solution is to completely replace the *NetworkManager* program transparently to the remaining desktop applications that make use of it, as is the case of the tray applets and the *nm-connection-editor* tool. This objective requires the implementation of a D-Bus interface with the same methods, properties and signals of the original *NetworkManager* program. This task is for the higher level component of the framework, described in section 5.6.

Concealed in this similar interface, however, is a very different solution, proposed in the previous chapter, and represented in Figure 5.1. The gray components were developed from scratch, except for the MIHF, which was only modified in order to support some extensions. In the lower layer of this solution are the Link SAPs, built on top of the standard Linux kernel interfaces and the most widely used core tools for network management. Section 5.4 describes the implementation of GNU/Linux Link SAPs for Wi-Fi and Ethernet interfaces.

The various components are developed using the C++ language; the ODTONE APIs, together with *boost*, already provide tools for marshalling MIH messages in the C++ language. Also, C++ programs can deliver a great performance and permit low level operations, while maintaining higher level features such as automatic memory management and the object-oriented paradigm (although it supports others).

Each component of this framework is effectively a separate process. Section 5.3 describes the interactions of some basic connection management procedures. These interactions rely on the Netlink protocol, for kernel operations, and on D-Bus, for receiving management instructions from the user and for controlling the *wpa_supplicant* daemon. The use of the Netlink protocol is explained in section 5.2. The integration of the D-Bus messaging framework is described in section 5.1.

Although *dhclient* does not offer any programmable interface, its support for a wider range of protocols makes it a better choice for a DHCP client. Its use is managed by scripted calls concealed in a C++ class, explained in the L3 Conf command description subsection (5.4.18).

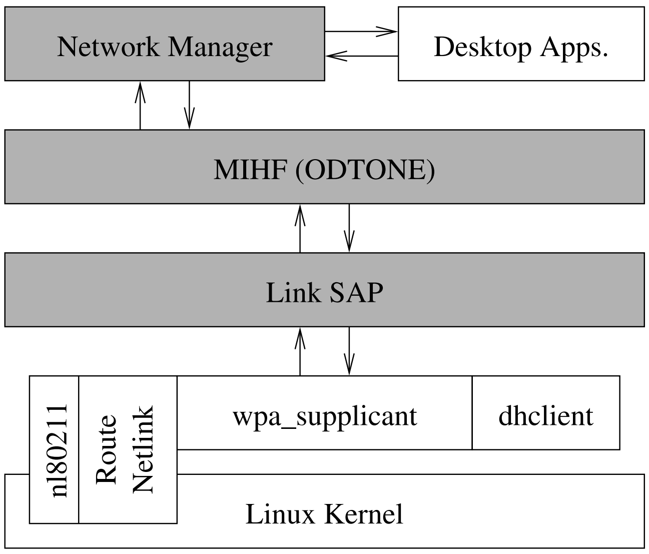


Figure 5.1: EMICOM architecture.

5.1 D-Bus integration

Both the MIH User and Link SAPs make use of the D-Bus framework: the MIH User, to export the NetworkManager interface, and the Link SAPs to make use of the *wpa_supplicant* program. There are two public C++ D-Bus binding implementations available: *dbus-cxx*[1] and *dbus-c++*[2]. Both are wrappers for the C library, but *dbus-cxx* aims at exposing the C API for direct manipulation ans uses *sigc++*[3] to provide an object-oriented interface for the framework. However, it does not implement mechanisms for mapping C++ class members into D-Bus interface properties, for example. *dbus-c++*, on the other hand, attempts to provide a complete object-oriented abstraction for the framework, including support for methods, signals, properties, and all D-Bus supported data types.

The D-Bus framework specifies an Extensible Markup Language (XML) data format for defining (and documenting) interfaces[4]. *dbus-c++* offers a tool for converting XML-defined interfaces into C++ classes for both server and proxy objects. Methods and Signals are mapped into class member functions, and Properties translate to class member variables. The inner working of each object may be implemented at these classes themselves, or by extending the class and implementing the respective methods. The wrapper code automatically created by the tool takes care of the message handling, and using D-Bus through these classes is transparent for the programmer. Messages from D-Bus, and ODTONE, are handled asynchronously, using the *io_service* dispatcher from the *boost Asio* libraries.

Despite *dbus-c++* being the obvious choice, it later presented some issues that had to be resolved. The current implementation did not support all the methods from the D-Bus

[1] *dbus-cxx*, `http://dbus-cxx.sourceforge.net/`

[2] *dbus-c++*, `https://gitorious.org/dbus-cplusplus`

[3] sigc++, `http://libsigc.sourceforge.net/`

[4] D-Bus introspection format, `http://dbus.freedesktop.org/doc/dbus-specification.html#introspection-format`

introspection interface, which would not allow third party developers to correctly introspect software using *dbus-c++*. It also failed to compile properly with recent versions of the GNU Compiler Collection (GCC)[1], and also performed incorrect message parsing in a specific case. These fixes were transmitted to the developer of the original library, and made available at the Code.UA[2] repository.

5.2 Route Netlink and nl80211 wrappers

Implementing a userspace MIH Link SAP means providing a mapping between each primitive of the MIH service and the kernel interface for the link layers. The Route Netlink and nl80211 interfaces are used for kernel communication, but the Netlink message handling is aided by the *libnl*[3] library. This library is developed in the C language and offers wrappers for the various Netlink protocols. The Route Netlink protocol is wrapped by the *libnl-route* component, and nl80211 can be accessed through the *libnl-genl* component. *libnl-route* provides a complete set of methods for Route Netlink object management, such that the NEWADDR message for adding an IP address to an interface can be handled in the following way:

```
1 struct rtnl_addr *addr = rtnl_addr_alloc(); // allocate message
2 rtnl_addr_set_ifindex(addr, ifindex);       // set interface
3 rtnl_addr_set_local(addr, local_addr);      // set address
4 rtnl_addr_set_family(addr, fam);            // set addr. family
5 rtnl_addr_add(socket, addr, flags);         // send message
6 rtnl_addr_put(addr);                        // deallocate message
```

These methods automatically create the message headers and the correct payload TLVs; lines 1 and 6 refer to explicit memory management.

The Generic Netlink support, though, does not provide an extensive set of methods to handle each message attribute. Instead, it provides methods that aid parsing and iterating over Generic Netlink payload values.

This library can be used directly in C++, given the compatibility with the C language. However, explicit memory management can be very problematic if not handled correctly. When returning or throwing from a method, all the explicitly allocated objects must be discarded. If many objects are allocated and the method has several points where it could return, the program becomes visually polluted with memory management procedures. Two solutions for easily writing safer code are described in Annex A. The Resource Acquisition Is Initialization (RAII) principle proved the simplest and the overall best approach, so every object or resource of the *libnl* library that was included in the project was wrapped in RAII containers to a component called *nlwrap*.

nlwrap provides a safer way to use the *libnl* library, but also takes care of message parsing for the required subset of the nl80211 protocol.

[1]GNU Compiler Collection, `http://gcc.gnu.org/`

[2]Code.UA software repository, `http://code.ua.pt/`

[3]*libnl*, `http://www.infradead.org/~tgr/libnl/`

5.3 Management Processes

Instead of the monolithic process that the *NetworkManager* program employs (apart from the integration with the authentication and dynamic IP configuration tools), EMICOM adopts a layered approach; this decoupled nature enables the desired OS and technology independence. This section is dedicated to the illustration of a few interactions between the various components, which could nonetheless represent the flows that are commonly concealed in monolithic solutions.

5.3.1 Startup

The execution of each component at the appropriate timing is aided by a shell script. In a correct startup, *wpa_ supplicant* must be running before the Links are launched; however, if it is not running yet, the D-Bus daemon should launch it when its interface is requested. If, at this point, the MIHF is not running yet, the discovery process fails, but the Link SAP is not terminated, as the MIHF can be statically configured with the address and port on which the Link SAP are listening. The MIH User, however, must only be launched once the MIHF has either statically or dynamically discovered the Links, so they will be detected via the Capability Discover message. A scenario with the dynamic discovery mechanism is shown in Figure 5.2. The available command line parameters for each component is available in Appendix B.

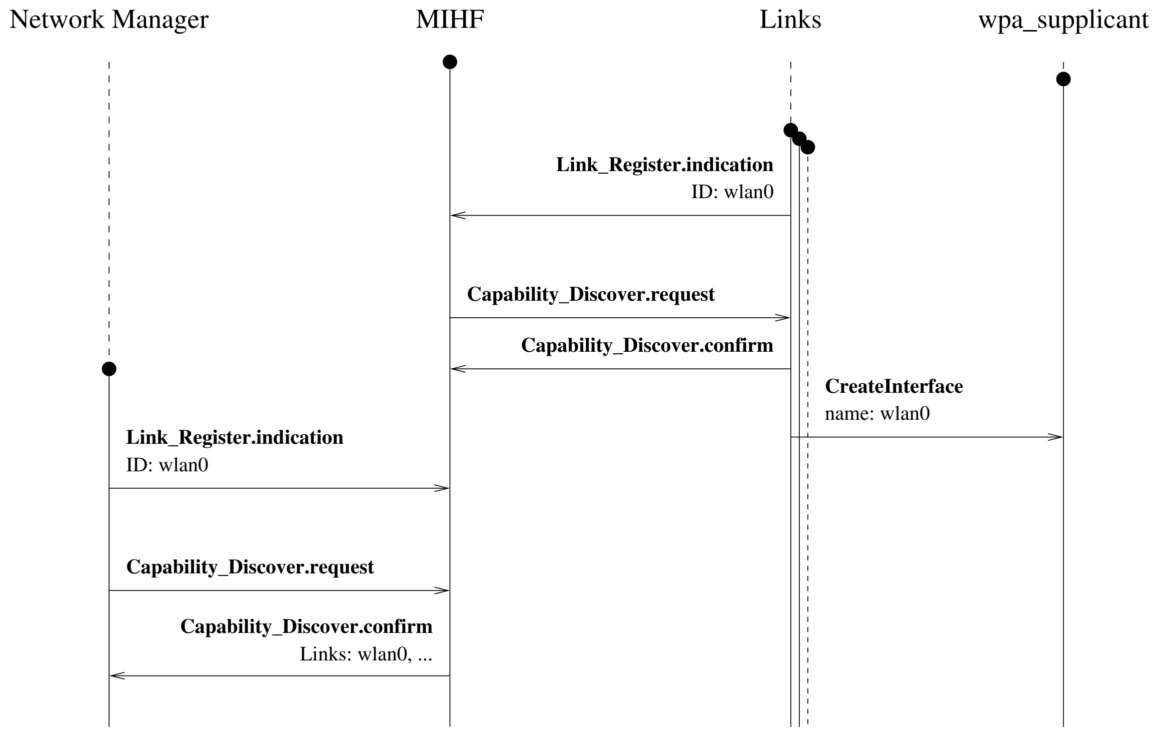


Figure 5.2: Message signalling for framework startup.

5.3.2 Connect

A successful connection to a network usually takes the steps depicted in Figure 5.3. It should be noted that usually, after a Link Up indication, it would be sensible to initiate the IP configuration procedure. However, this is not carried out until the Link Conf request is confirmed as successful. The reason for this is that, although a Link Up indication occurs right after the authentication and association steps, this does not mean the security procedures are completed, and data other than EAP frames, for example, could be blocked if the network requires 802.1X authentication.

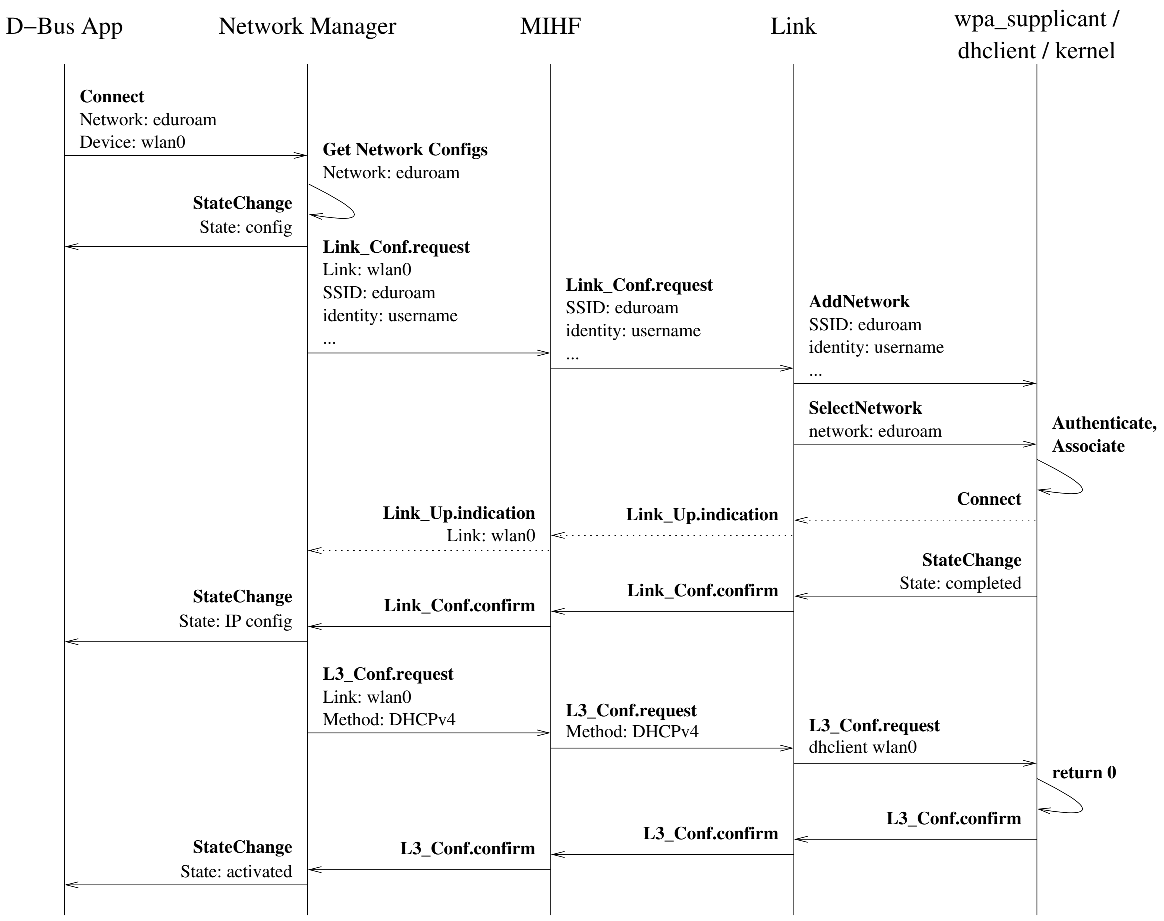

D–Bus App
Network Manager
MIHF
Link
wpa_supplicant / dhclient / kernel
Connect Network: eduroam Device: wlan0
Get Network Configs Network: eduroam
StateChange State: config
Link_Conf.request Link: wlan0 SSID: eduroam identity: username ...
Link_Conf.request SSID: eduroam identity: username ...
AddNetwork SSID: eduroam identity: username ...
SelectNetwork network: eduroam
Authenticate, Associate
Connect
Link_Up.indication Link: wlan0
Link_Up.indication
StateChange State: completed
Link_Conf.confirm
Link_Conf.confirm
StateChange State: IP config
L3_Conf.request Link: wlan0 Method: DHCPv4
L3_Conf.request Method: DHCPv4
L3_Conf.request dhclient wlan0
return 0
L3_Conf.confirm
L3_Conf.confirm
L3_Conf.confirm
StateChange State: activated

Figure 5.3: Message signalling for a connection request.

5.3.3 Disconnect

Network disconnection, from the MIH User's perspective, occurs in a single step, shown in Figure 5.4. The MIH Link Actions disconnect command in this framework takes the meaning of not only disassociating the current L2 link, but also clear the configured network layer attributes of the interface.

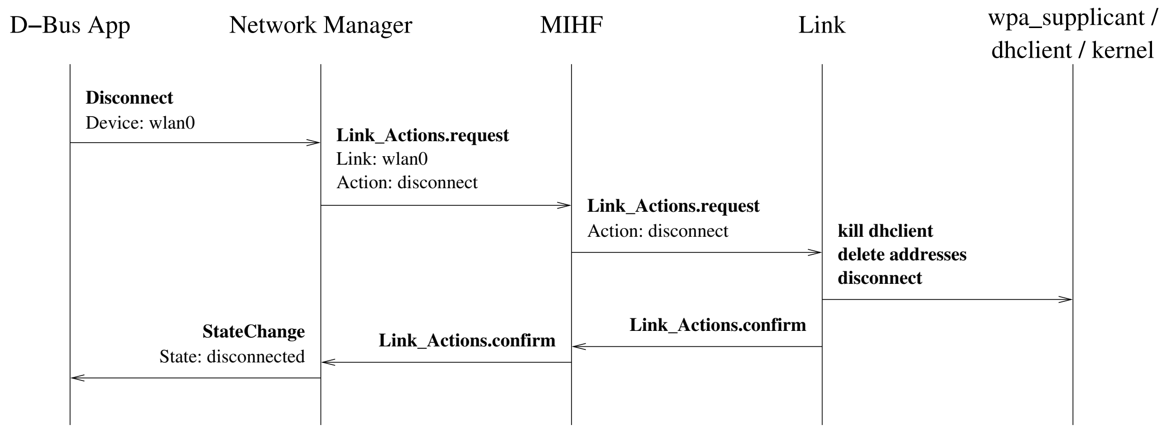


Figure 5.4: Message signalling for a disconnection request.

5.4 Link SAPs implementation

The main structuring of the Link SAP implementations for GNU/Linux is represented in Figure 5.5. For the most part, an 802.3 Link SAP is a subset of the implementation for the 802.11 counterpart. The following sections cover the initial setup procedures for the Link SAPs, and the mappings for each primitive of the 802.11 Link, referencing differences for the 802.3 implementation where appropriate.

5.4.1 Initial Setup

The Link SAPs are launched from command line, at userspace, but require super user privileges. Upon launch, a privilege check is performed first. If no super user privileges are granted, a warning message is issued, but the process continues, as event functionality is still supported without these privileges.

After the initial privilege check, the configuration parameters are parsed from the configuration file and the command line switches, using the Program Options library from *boost*. Command line switches have precedence over options indicated by the configuration file.

Once the configurations are parsed, the services are initialized:

- The supported command and event list are set;
- The Link SAP registers with the MIHF;

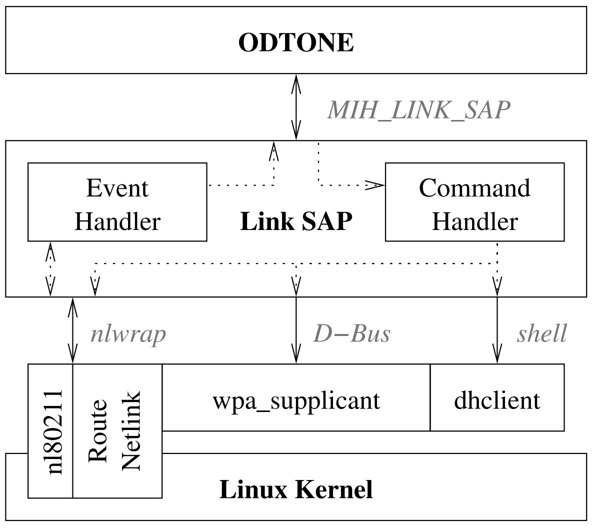


Figure 5.5: Link SAP implementation diagram.

- Several nl80211 and Route Netlink multicast groups are subscribed, for event notifications;
- The D-Bus client proxies are started, and the respective interface is configured in the *wpa_ supplicant* daemon;
- The program enters an infinite loop, dispatching D-Bus and MIH tasks.

When a message is received at the Link SAP, from either the MIHF (Commands) or the kernel (Events), the appropriate procedures are handled as explained in the following sections.

5.4.2 Link Detected event

This primitive is generated by the Link SAP every time a new PoA of an access network is detected, and provides information with a LINK_DET_INFO payload. This primitive does not apply to the 802.3 standard. For the 802.11 technology, it is generated whenever a scan detects new APs. This occurs when one of NEW_SCAN_RESULTS, SCAN_ABORTED or SCHED_SCAN_RESULTS messages is propagated on the nl80211 *scan* multicast group. These messages only inform of the scan completion; after these events, detailed scan results are obtained by sending a GET_SCAN command to the kernel. This response contains some basic TLVs, the most relevant being the BSS_INFORMATION_ELEMENTS TLV, which contains the beacon IEs encoded as defined in the 802.11 [10] and 802.11u [54] standards. The fields for the Link_Detected message are mapped according to the following list:

- **Link Tuple ID**: this field identifies the detected PoA via its link layer address. The AP's MAC address is contained in the BSS_BSSID TLV of the GET_SCAN response.
- **Network ID**: for 802.11 networks, this is the SSID, which is the first defined IE.

- **Network Auxiliar ID**: this is the Homogeneous ESS Identifier (HESSID), contained in the "interworking" beacon IE, only defined in the 802.11u draft.
- **Signal Strength**: the BSS_SIGNAL_MBM TLV contains the beacon energy in $100 \times dBm$ units.
- **Signal over Interference plus Noise Ratio (SINR)**: this field cannot be set, as nl80211 does not report interference or noise information.
- **Data Rate**: the maximum data rate for an AP is the highest value reported in either the "supported rates" or "extended supported rates" IE.
- **MIH Capability Flags**: this field indicates which MIH services are supported by the PoA. The advertisement protocol IE contains this information. The IE has one identifier for the information services support, and another identifier for both command and event service.
- **Network Capabilities**: this is a collection of flags indicating whether the PoA supports any security features, the available QoS classes, if it provides internet access, emergency services, and MIH capability. These fields are extracted from specific IEs, most defined in the new 802.11u draft.

Scans must be initiated by applications, as the kernel takes no action in this regard (but specific drivers might). The MIH Users are able to request scans through the MIHF, but this would cause unnecessary recurring message exchanges. The developed Link SAP can be configured to perform periodic scans, by indicating a trigger period in ms. Instead of using the scheduled scan feature of the kernel, the Link SAP will control the timer internally, which yields greater control over concurrent operations.

The MIH standard specifies that only the first detected PoA of a network should be announced. If more than one AP of the same network is detected in the same scan result, the Link SAP will assume the one with the stronger signal is the one to announce, as it is most likely the first encountered.

The event will be propagated to the MIHF if it was not previously subscribed.

5.4.3 Link Up event

The Link SAP will generate this event every time the interface acquires link layer connectivity. This occurs after the authentication and association steps, and indicates the ability to send link layer frames, so it occurs before the setup of security features by supplicant daemons. The nl80211 *mlme* multicast group informs the association event in the form of CONNECT commands, which translate to the following Link_Up message fields:

- **Link Tuple ID**: this field identifies the local link type and address, both configured when the Link SAP is launched, and includes the newly attached AP's MAC address as well, which is provided by the BSS_BSSID TLV.

- **Old Access Router address** (optional): the old access router is not indicated.
- **New Access Router address** (optional): the new access router is indicated as being the AP's MAC address.
- **IP Renewal Flag**: indicating this field is not supported, as the CONNECT event does not provide layer 3 parameters.
- **IP Mobility Management**: this field is not indicated, for the same reason as the Renewal Flag.

The CONNECT command does not always indicate a connection success. The same command is used following a failed association attempt. The STATUS_CODE TLV indicates the success of the operation; the Link_Up event is only generated when this code indicates a successful operation.

For IEEE 802.3 devices, the link layer status can be determined via the operational state object from the Interfaces Group MIB [55]. Changes to the operational state of an Ethernet device are propagated in the Route Netlink protocol, and the operational state "up" indicates that link layer frames may be transmitted. The Link_Up event for Ethernet devices includes only the Link Tuple ID, with local interface information.

5.4.4 Link Down event

The nl80211 *mlme* multicast group propagates a DISCONNECT command whenever link layer connectivity is lost. The loss of link layer connectivity for wireless devices can only be detected if explicitly requested, otherwise it is based on metrics such as the consecutive failure in transmission or reception from a PoA. This mechanism is handled at the kernel, by the drivers themselves. A Link_Down message is generated for every DISCONNECT command, as follows:

- **Link Tuple ID**: obtained similarly to the previous messages.
- **Old Access Router address**: not supported.
- **Link Down Reason**: converted from the REASON_CODE TLV in the DISCONNECT message.

Similarly to the Link_Up command, the indication for 802.3 devices is done via LINK messages indicating a change to the operational state object.

5.4.5 Link Parameters Report event

This message is used by the event service to send various metrics periodically, or if they cross a given value. The following metrics are described in the protocol:

- **RSSI** (802.11): no nl82011 command reports the RSSI *per se*, and Linux interprets this value as the current measured signal energy, in *dBm*. In reality, the 802.21 also means this interpretation, as it does not require a mechanism of retrieving the device's RSSI_Maximum, thus rendering an arbitrary RSSI value meaningless. The nl80211 interface offers one method for retrieving various connection parameters against a given 802.11 network device: the GET_STATION command. The response contains information and statistics about an associated station which, in the infrastructure mode, corresponds to the Access Point. The INFO_SIGNAL attribute from a GET_STATION response always corresponds to the perceived strength of the last received PDU from the AP. This value has a great variability; an INFO_SIGNAL_AVG attribute provides an average of the signal, but is not used in this case.
- **Multicast Packet Loss Rate** (802.11): this metric is not available, as there is no specific statistic for multicast traffic.
- **Data Rate**: the data rate for 802.11 devices is also obtained via the GET_STATION command. The Route Netlink protocol does no offer data rate information for any devices. For Ethernet devices, this is only available via *ioctl* call or by reading the speed attribute from the network device object in the kernel, via *sysfs*. The *ioctl* call is used for this parameter, due to uncertainties regarding the *sysfs* path stability across systems for this file.
- **Packet Error Rate**: using Route Netlink, it is possible to retrieve an extensive count of sent and received packets, including error counts. The combination of these statistics is used to calculate the packet error rate at a given time.
- **Signal Strength**: this parameter is interpreted similarly to the Wi-Fi RSSI parameter.
- **SINR**: as explained in the Link Detected message (5.4.2), this parameter is not supported.
- **Throughput**: the statistics used to calculate the Packet Error Rate feature can also be used to track interface-global transmission throughput.

Currently, there are no configurable thresholds for the Ethernet link type. An additional class of parameters are available in the 802.21 standard, for retrieving QoS statistics per Class of Service (CoS), such as the minimum, maximum and average packet delay. However, the Linux kernel does not provide this kind of statistics. Supporting them would require changing kernel code for the packet schedulers, for example.

5.4.6 Link Going Down event

According to the 802.21 standard, this event can be triggered by well known procedures, or by heuristic approaches to varying conditions, to indicate that link layer connectivity is expected to be lost in a given time frame.In Linux, whether the kernel or a userspace routine causes an interface to disconnect, there is no standardized indication of the occurrence until it effectively occurs. Hence, this event cannot be directly supported.

An heuristic approach requires monitoring various link parameters, and there is no single solution for the problem of determining the connectivity loss, as there is no way of determining the time frame for occurrence of a connectivity loss event. Most algorithms are based on metrics that are available to the userspace via the MIH event and command services, so the prediction of connectivity loss events can still be supported by MIH Users.

As such, this event is not supported in the developed Link SAPs. This event should reflect kernel or network-side management operations advertised to the connected peers, which are not yet supported in the standards.

5.4.7 Link Handover Imminent event

This event is not implemented. Supporting this event faces a problem similar to the one described for the Link Going Down (5.4.6). That is, when a link layer decision of handing over a connection is made, there is no indication of the occurrence to userspace applications until the operation is completed, so there is no way of telling that a link layer handover is about to occur.

5.4.8 Link Handover Complete event

The Link Handover Complete event indicates a link layer handover completion. Unlike the Link Handover Imminent event described previously, it can be detected via the features provided by nl80211. The kernel uses the ROAM message to indicate that a link layer handover was performed between BSSs. However, this is not the most commonly seen behavior, and it is more common to see a DISCONNECT followed by a CONNECT event sequence. Since the DISCONNECT message will immediately generate a Link_Down event, the Link_Handover_Complete is only generated if the ROAM message is used.

5.4.9 Link PDU Transmit Status event

This mechanism, as explained in the 802.21 standard, requires implementing a layer that mediates the transmission of all network packets, and further requires higher layers to indicate a unique identifier per packet. The developed Link SAP runs at userspace; intercepting packets would imply a great overhead, and this feature would not be useful for the time being, so it was not implemented.

5.4.10 Link Conf Required event

This message follows an attempt at associating to a network, which requires further elements in order to complete the setup procedures for link layer connectivity. This message is generated whenever *wpa_supplicant* issues a NetworkRequest D-Bus signal. Each signal contains a pair of strings, indicating the field type and description for the requested parameters. Both parameters are propagated in the Event to the MIHF.

5.4.11 Link Capability Discover command

The set of supported events and commands for each Link SAP is statically known for each implementation. Currently, the response from this command is also static.

The nl80211 interface supports querying links in order to determine their supported commands and events as well. However, has all hardware seemingly supports the necessary features, the supported services are not dynamically discovered. If it is the case that some hardware does not support certain required features, the dynamic discovery method must be implemented, following an algorithm such as this, to be executed during the setup phase:

1. Start with a complete list of 802.21 primitives, and the required nl80211 commands and events for each.

2. Query the device capabilities via nl80211.

3. For each 802.21 primitive, see if the required nl80211 messages are supported. If at least one command or event is missing, remove the primitive from the capability list.

Newer Linux kernel versions might also provide additional support for certain features that cannot be implemented in the Link AP yet. However, until those features are released, there is no way of knowing whether or not they are supported without actually programming the support for them.

5.4.12 Link Event Subscribe command

The MIHF delivers event subscription commands on behalf of MIH Users. The ODTONE implementation in particular takes action in order to reduce message exchanges between entities: if an MIH User requests a list of events that has already been subscribed for that Link SAP, the request is not forwarded to the destination and the MIH User immediately receives a success response. If the request contains at least one event that has not been subscribed, the MIHF redirects the request to the intended Link SAP. This redirected request contains the complete list of subscribed events; from the perspective of the MIHF, and resource management, there is no benefit in stripping the list before sending it to the Link SAP, as the list of events is a fixed-size bitmap.

The list of subscribed events is stored in each Link SAP, in the form of flags. During startup, the flags indicate that no event is subscribed. Over time, with the reception of Link Event Subscribe commands, the requested list is first stripped against the list of supported events, to remove unsupported events, and then it is added to the list of subscribed events.

The response to a Link Event Subscribe command will always include the list of requested events that are supported as successfully subscribed, even if they have been previously subscribed. Similarly, the MIHF will redirect the complete response to the source MIH User, so it will never get an error response for subscribing the same events twice (which would result in uncertainty about whether the subscription went wrong or the subscription just would not make sense).

5.4.13 Link Event Unsubscribe command

Unsubscribing events comprises a series of operations very similar to the event subscription counterpart, described in the previous command: a received list of events for subscription is first stripped to remove the unsupported events, then the list of subscribed events is updated. No errors will arise from unsubscribing already unsubscribed events.

5.4.14 Link Get Parameters command

The response to a Link Get Parameters command is very similar to the Link Parameters Report event, the greatest difference being in respect to how the parameters report are configured. A Parameters Report may be issued periodically or when a threshold is crossed, whilst a Get Parameter command will retrieve the given value immediately. The list for supported parameters is the same as for the Parameters Report event (see 5.4.5).

Additionally, the Get Parameters command can be used to get the current device operational mode and the channel in use, for wireless links. Determining whether a link is powered on or off is done via the Route Netlink GET_LINK message and checking the respective flags. Wi-Fi devices can also be queried for their power saving status via the nl80211 GET_POWER_SAVE command. The frequency channel for a link can be obtained via the GET_STATION nl80211 command but requesting this parameter when there is no active connection will cause a failure response.

5.4.15 Link Configure Thresholds command

The threshold configuration is processed according to the following sequence, also described in Figure 5.6:

1. For each requested parameter:

    a) check whether it is supported. If not, set a failure status for the specific parameter and move to the next parameter.

    b) If the parameter is supported, the attributes of the threshold configuration request are interpreted:

        i. If the “cancel” action is requested and no list of threshold values are given, all previously configured thresholds for that type are removed. If threshold values are indicated, only thresholds with the specified values are canceled. If a cancel is requested for a threshold that was not configured, no error message is generated.

ii. If a “period” is indicated, it is regarded as a request for a periodic report for a given value. This is attributed it’s own thread, that wakes up at the requested interval to report that value.

iii. If the “one shot” attribute is set, the threshold values are inserted in a list, with an indication that the specific threshold is to be removed once it is crossed.

iv. The “normal” action is similar to the “one shot”, except the given threshold will be continually verified until it is explicitly canceled.

2. After finishing the configuration of the thresholds, the threads for verifying the thresholds are stopped/started, depending on the resulting configuration.

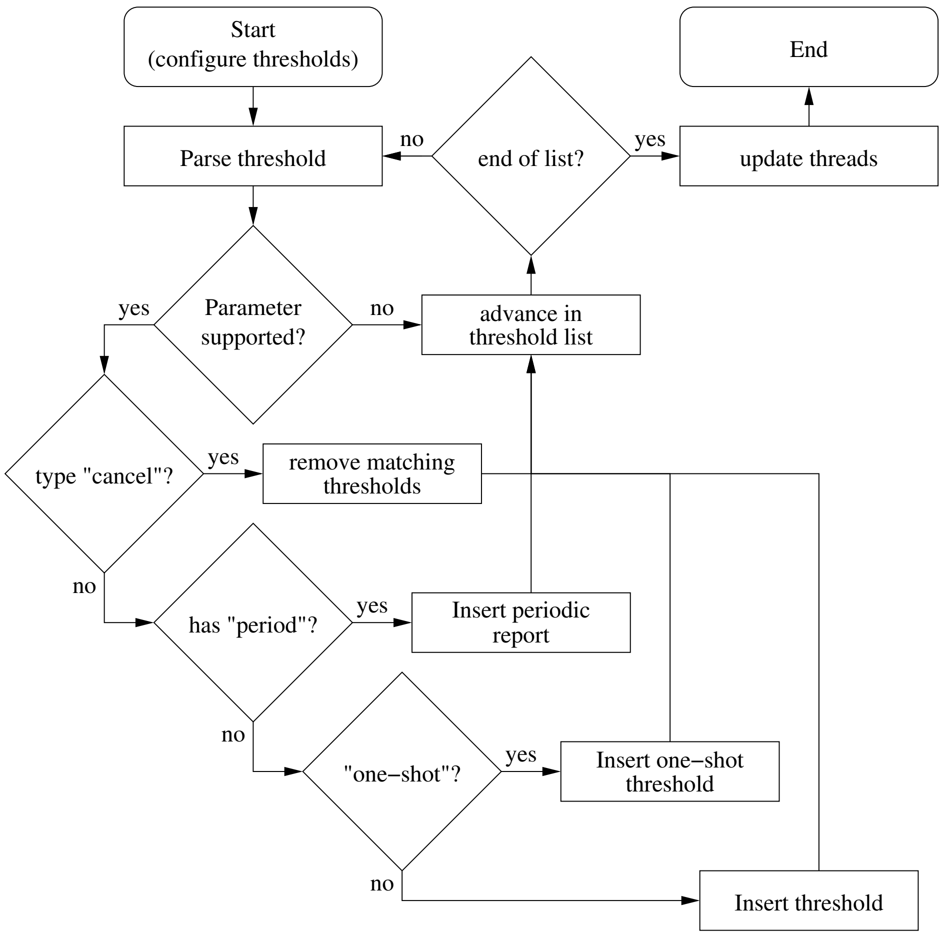


Figure 5.6: Threshold configuration command handling.

The nl80211 interface supports configuring thresholds at the kernel level for variables such as signal strength and packet error rates. However, this mechanism only allows configuring one value. The 802.21 standard allows configuring various thresholds on the same parameter so, for example, it should possible to subscribe notifications for both when the signal strength goes below $-80dBm$ and above $-20dBm$. The need to support this flexibility (and the fact that it does not support as many parameters or even Ethernet devices) led to ignoring the threshold configuration feature of the nl80211, and performing all threshold checks by periodic polling. A configuration parameter for the Link SAP allows defining the default period for threshold checking, in milliseconds.

The parameter reporting features apply only to connected links. When there is no connectivity (on Link Down event), or the device is powered off, the threads for threshold verification and parameter reporting are deactivated until the link comes back up (on Link Up event).

5.4.16 Link Actions command

The Link Actions command is used to control the operational state of the device. Powering the interface up and down is done via the Route Netlink protocol by respectively setting or unsetting the IFF_UP flag in a LINK message. The request to disconnect an Ethernet interface is equivalent to powering down the device. Wi-Fi devices are disconnected by issuing a DISCONNECT command on nl80211. Setting the device to low power is translated into enabled the power saving feature via the SET_POWER_SAVE command (not supported in Route Netlink, for Ethernet devices).

The Link Actions messages also allow requesting a device to scan. If a scan is requested and the device is powered down, an error message will occur. It is also common that a scan request will be only partially completed if a connection is active, in which case the result will contain an error indication. The process of obtaining the scan results is similar to the Link Detected event generation (5.4.2), although a response for the scan request contains less information than a Link Detected message (although completing a scan will separately trigger Link Detected messages).

The response to a Link Action message with a scan request may be generated only after a few seconds, depending on the time the device takes to complete the scan operation. Furthermore, the command may indicate a delay time, in milliseconds, for execution of the action, which is enforced before any action is performed.

Another parameter of the Link Actions primitive can request the Link to retain resources on disconnecting, so that a later link connection can be more efficient, as well as requesting the serving PoA to forward buffered data to the new target PoA, preceding a handover attempt. These features refer to handover management in mobility scenarios, which are not yet considered, and thus not implemented.

5.4.17 Link Conf command

The Link Conf command is meant as a request to authenticate/associate to a network, including the setup of required security procedures. Since this command was developed with the integration with *wpa_supplicant* in mind, the provided configuration elements are directly related to the *wpa_supplicant* key-value pairs for network configuration. Although the message format for the Link Conf message is based on a list of OCTET_STRING pairs, some parameters have to be converted to different types before communication with the *wpa_supplicant* daemon. Thus, receiving this command automatically translates to the following requests on the D-Bus Interface object of the *wpa_supplicant* daemon:

1. **AddNetwork**: this method accepts a *wpa_ supplicant* network configuration block that is added to the list of the daemon's known networks. This step takes place even if there are no security procedures, in which case the network configuration contains only the desired SSID and other *wpa_ supplicant*-specific configurations

2. **SelectNetwork**: this is used to effectively request the *wpa_ supplicant* to initiate the authentication, association and security procedures for the configured network.

wpa_ supplicant propagates D-Bus signals announcing device state changes. A change to the "completed" state indicates that the configuration procedures were successfully completed, whereas the "disconnected" mode indicates a failure in setting up a connection. The Link SAP waits for these signals in order to include the status information in the response message.

The D-Bus interface will also signal network requests for missing security tokens that the supplicant must provide. Receiving this signal while attempting to configure a network means that the particular operation was not successful, but does not imply that the connection failed. It could mean that further steps are required from the Mobile Node or the user. This special case will issue an "authorization failure" (as opposed to a simple "failure") MIH response, followed by an MIH Link Conf Required event.

5.4.18 L3 Conf command

Several mechanisms are employed for IP and DNS configuration. It is possible to request the Link SAP to perform dynamic IP configuration, using DHCP, and, at the same time, provide additional addresses, routes or DNS servers, without conflict.

DHCP configuration is accomplished using *dhclient*, referred in 3.3.5. It was selected over *dhcpcd* for being developed by ISC and providing DHCPv6 support. However, as it does not expose an API, a wrapper was developed for scripted invocation of the daemon. This wrapper provides methods for forking the process with various options, and saves its PID, for future cancellation when a DHCP release is required. Via command line options, the *dhclient* program can be requested to perform DHCPv4 and DHCPv6, as well as stateless IPv6 autoconfiguration (by performing Information requests for neighbor discovery).

After dynamic configurations, if the request contains IP addresses or routes to configure, they are associated to the network interface via the Route Netlink protocol using ADDR and ROUTE messages.

Manual DNS configuration is achieved by directly writing to the DNS configuration file for GNU/Linux systems. This file has support for many options such as request timeouts, number of attempts and server list sorting, but currently only name server entries are added.

A new L3 Conf request will always clear the previous network address configurations, as will the Link Actions "disconnect" or "power down" request commands.

5.5 MIHF Extension

The ODTONE library implements the original 802.21 primitives. The additional Link Conf, L3 Conf and Link Conf Required detailed in the previous section are implemented in the Command and Event service similarly to the existing primitives.

First, a Message ID code is assigned to each primitive, which is used in the message headers. The attribution follows the guidelines from the 802.21 standard, and does not cause inconsistencies with the existing messages, although identifiers for other existing extensions might collide with the attributed codes. Additional message attributes for network identification and security parameters interchangeable with the *wpa_supplicant* and *NetworkManager* formats are also proposed, and the respective TLV encodings are defined once again following the reserved codes for extensions.

The handlers that process each message to and from the Links and MIH Users are implemented in the command and event service ODTONE components, alongside the existing handlers and code structure.

There is an ODTONE parameter for specifying the response timeout for MIH commands. This value is to be made small enough that entities can perceive failures quickly. However, tasks such as security setup and IP negotiation can potentially take several seconds. As such, the timeout handlers for the Link_Conf and L3_Conf primitives are made to be triggered after their own timeouts, independently configured before launch.

All the code affecting the ODTONE library is enclosed in a specific pre-processor macro that helps users compiling it without support for the primitives.

5.6 Network Manager implementation

The mechanisms described in this section further evidence the need for various commands and mechanisms implemented in the Link SAP, including the proposed MIH extensions.

The main goal of the MIH User was to expose the functionality provided by the existing GNU/Linux *NetworkManager*, by implementing the same D-Bus API. Desktop applications use the API for enabling and disabling network devices and connections, and giving visual feedback about the current network state. Hence, in its current state, EMICOM does not employ automatic decision algorithms, and implements only functionality to respond to user inputs via the D-Bus API and the various tools that use it. In this way, the framework does not enforce a specific method for conectivity management.

This section also highlights certain *NetworkManager* features that are not supported, mostly due to missing support from the MIH primitives. Extensions to support those features are not implemented, since they were deemed not crucial for network managing operations.

D-Bus, and the NetworkManager API in particular, promotes object-oriented program designs. The implemented architecture is represented in Figure 5.7, with five major components following the D-Bus architecture for the NetworkManager interface. Each component is explained in the following sections.

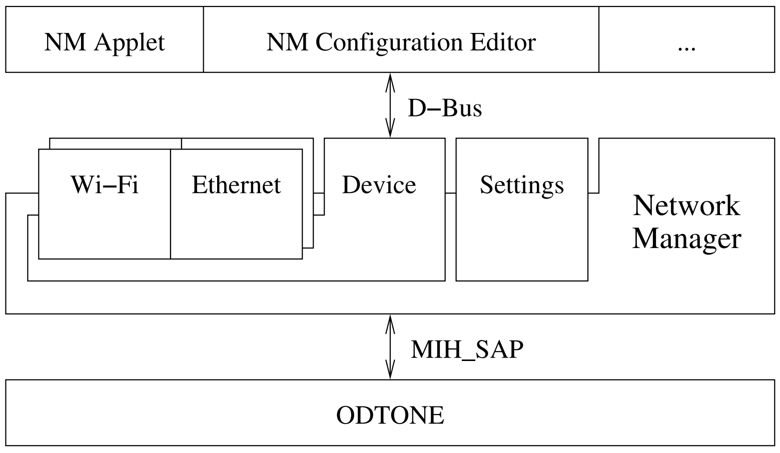


Figure 5.7: Network Manager MIH User components.

5.6.1 Network Manager

The Network Manager component is the central object that manages the other entities, and provides the basic interface with both the desktop applications and the MIH services. Upon launch, this object initiates the discovery process via the MIH Command Service by registering itself as an MIH entity and requesting information about existing Links and their capabilities. Currently, all available Links have to be registered with the MIHF before the MIH User is launched; hotplug support would require the MIHF to announce newly registered Links, via the Event Service, or that the MIH User kept polling for new Links via the Command Service.

After the discovery process, a Device object is created for each discovered Link, according to its specific link-type field (for example, an 802.11 Link will trigger the creation of a Device Wireless object). Some events are also immediately subscribed, which will enable monitoring the Link's state (signal strength and data rate). Events pertaining to device states are not delivered directly from the Event Service to the Device object. Instead, they are first processed by the Network Manager object in order to control connectivity and status update operations.

This interface allows globally enabling or disabling network management, as well as initiating or terminating connections on specific devices. Initiating a connection via the D-Bus interface implies indicating the required network configuration, stored in the Settings component, and the device on which to initiate the connection. It is possible to initiate a connection without referring the configuration object, in which case the user provides a reference to an element that will help create the network configuration on demand (for example, a reference to the desired AccessPoint object helps filling the network configuration parameters for an 802.11 network).

5.6.2 Settings

A single Settings object is launched for the lifetime of the MIH User. The interface exposed by this object supports the same Ethernet and Wireless network configuration use-cases as the original *NetworkManager* program, using the *nm-connection-editor* tool previously shown in Figure 3.7.

Each network configuration is associated with a unique Connection object, which uses an internal data structure for conversion both to the *NetworkManager* settings map format, used by the *nm-connection-editor* tool, and to the 802.21 format for interaction with the Command Service. The Connection object class also provides a marshalling and unmarshalling method for data persistence. Configurations are stored in a directory, where a file will be created for each Connection object. The *boost* PropertyTree library allows storing and parsing information from configuration files in the INFO, JSON, XML and INI formats. However, since there is no intention of having users directly editing the files, as the *nm-connection-editor* tool provides access to all variables, persistence is accomplished using object serialization using the *boost* Serialization library. Not only this method provides greater space-efficiency, it is also easier and more efficient to parse than other human-readable formats.

Currently, there is no support for VPN, mobile broadband, Digital Subscriber Line (DSL) modems or Bluetooth features of the *nm-connection-editor* tool.

5.6.3 Device

The Device class is the base data type for all device types. Each Device object holds a reference to the MIH Command Service that it uses to perform control on the underlying Link. This includes the methods to Enable, Disable or Disconnect the device, which are translated to Link Action MIH messages, but also the link and network layer configuration.

The Device interface emits D-Bus notifications on device state changes, signalled by Link Up and Link Down MIH messages. However, as mentioned earlier in 5.6.1, MIH event messages are not subscribed directly by Device objects; they are first processed and multiplexed by the NetworkManager component, which then indicates each Device of the occurrence. Other than Link Up and Down, the NetworkManager also updates the Device for the various states of the connection process, as described in the Device D-Bus interface (3.4.2.2).

5.6.4 DeviceWired (Ethernet)

The DeviceWired interface extends the base Device class to provide additional properties and functions, as described in 3.4.2.3. The boolean Carrier attribute, which indicates whether or not a cable is attached to the interface, is not supported, as there is no MIH mapping for the parameter. The lack of support for this attribute causes no malfunction, but interfaces might rely on it to prevent users from requesting a connection on an Ethernet interface that has no cable attached, and will inevitably fail. Figure 5.8 evidences this type of feedback.

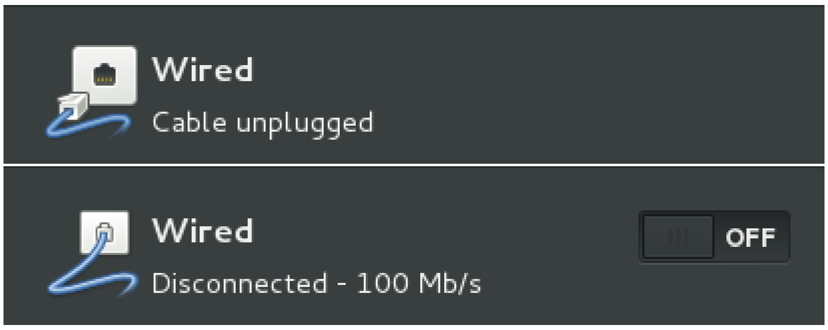

Wired
Cable unplugged
Wired
Disconnected - 100 Mb/s

Figure 5.8: Example of carrier detection feedback.

Another unsupported feature is the virtual HardwareAddress. It is always equal to the PermanentHardwareAddress, and there is no support for changing the HardwareAddress of an interface through the EMICOM framework. In case such a feature would be required, it could be implemented via an additional parameter in the Link Conf command.

The DeviceWired interface generates D-Bus signals whenever there is an attribute change, which are used mostly by the desktop applets and notification systems.

5.6.5 DeviceWireless (Wi-Fi)

Similarly to the DeviceWired interface the DeviceWireless interface does not support the virtual HardwareAddress attribute. Attribute changes will trigger D-Bus signals as well. DeviceWireless interfaces implement an additional method, for requesting radio scans.

This object also exposes a list of available APs, which is maintained using the information from the Link Detected events. If the Device stops receiving updates from certain previously added AccessPoint instances, they are removed after a given timeout. This means that until a timeout occurs for one AccessPoint object, it may be the case that the particular AP is actually no longer reachable if a user requests connection to it. Applications should minimize this risk (also present in any other Network Manager) by requesting scans immediately before requesting a network connection, for example.

5.6.5.1 AccessPoint

The AccessPoint interface expects many scan results information for each object. Most of it is supported. However, certain attributes are also missing due to lacking MIH primitive attributes.

The missing attributes refer to the security capabilities of each AP, namely whether it supports WEP, WPA, Counter Cipher Mode with Block Chaining Message Authentication Code Protocol (CCMP), TKIP, 802.1X, etc. These features are available at the kernel interfaces; however, the MIH the Link Detected event message only indicates whether or not security is supported, ignoring individual mechanisms.

The lack of this parameter results in difficulty for the user in setting up the configuration for unknown networks. Nonetheless, when the user attempts a connection to a network with all the required information, the Link Conf Required message will indicate the missing fields for authentication, which may then be introduced.

5.7 Summary

The developed framework implements the basic functionality for connection management. The existing *NetworkManager* applets and configuration tools can be used transparently to the user. Some parameters of the old D-Bus are not supported, but they were not implemented since they do not limit basic network management.

This architecture supports the common tasks offered by most connection management solutions. The deployment of a 802.21 infrastructure in the networks will effectively enable advanced connection management, encompassing policies traditionally not available for network selection algorithms. The available GUIs, and the D-Bus interface, should be updated to reflect these new capabilities.

Chapter 6

Evaluation

The success of the developed architecture is analyzed in terms of impact over the existing *NetworkManager* solution. Since the developed framework consists of a multi-process solution that relies on IPC mechanisms, there is an obvious overhead in communication. In the following sections the EMICOM framework is evaluated in terms of system footprint, and compared with the existing GNU/Linux *NetworkManager* in aspects such as the amount of code, memory consumption and battery drainage, which are important in mobile devices.

The benefit of enhanced network management comes from the IEEE 802.21 standard. Section 6.5 gives a view of various different scenarios that distinguish the EMICOM framework from common network management solutions, made possible by the mechanisms introduced with the MIH architecture.

6.1 Test Setup

The tests were run in a laptop computer with the specifications defined in Table 6.1. The testbed for network experiments contains two wireless Linksys[1] WRT54G AP with the *DD-WRT*[2] firmware, connected by Ethernet to a video server machine, as depicted in Figure 6.1.

Component	Value
Operating System	Archlinux
Kernel version	3.5.4
Processor	Intel Core i7 M620 ($2 \times 2.67GHz$)
Memory	$4GB$ at $1333MHz$
Ethernet card	Intel PRO/1000 CT
Wi-Fi card #1	Intel Centrino Advanced-N 6200
Wi-Fi card #2	ASUSTeK WL-167g

Table 6.1: Computer attributes.

[1]Linksys, `http://home.cisco.com/`

[2]*DD-WRT* firmware, `http://www.dd-wrt.com/`

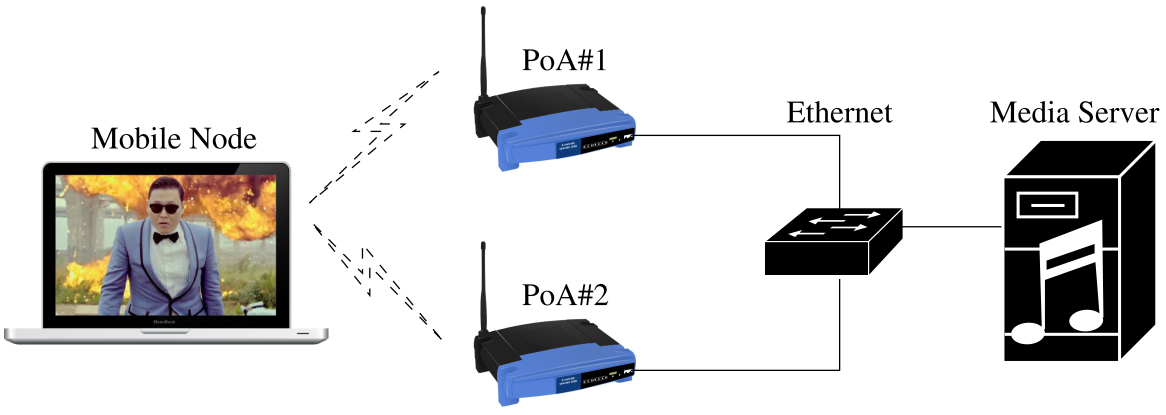


Figure 6.1: Testbed architecture.

6.2 Inter process overhead

In practice, the proposed solution adds a layer to the existing kernel interfaces for network management, which abstracts the media dependent control. Communicating through this layer introduces an overhead that causes delays between operations, and implies data transmission between processes, at a cost.

The 802.21 standard defines data transmission between remote entities via transport protocols such as UDP and TCP, encoding the necessary information in the TLV format. ODTONE uses this format for local transmission over transport sockets as well, allowing it to achieve the ability of being OS-independent, but at an obvious cost. Communication between the higher and lower layers require transmitting from the MIH User to the MIHF, then from the MIHF to the Links; answers traverse the inverse path. Messages from the event service travel only from the Link SAP to the MIH User, via the MIHF as well. Communication with remote entities is a necessary overhead and does not account for IPC analysis. Table 6.2 shows the number of MIH messages exchanged for various situations, including the total payload of transmitted data.

Operation	# of Messages	Total payload (bytes)
Power DOWN	4	162
Power UP	4	164
L2 Connect (simple)	4	197
L2 Connect (WPA + EAP)	4	508
L3 Configure (DHCP)	4	193
Disconnect	4	162
Link Detected event	2	144
Link Down event	2	80
Link Up event	2	102

Table 6.2: MIH message sizes.

From these values it is visible that Link events take the fewest amount of bytes required (between 80 and 144 bytes). Link commands demand more information (between 162 and 197 bytes), but it is the new more complex commands, where the DHCP and security association and capabilities have been added to the standard 802.21 behaviour, that require the most amount of information (between 193 and 508 bytes, respectively).

The delay for the transmission of these messages is not analyzed, since the performance is internal and highly dependent on the load and capacity of each system. However, [56] shows that, for message sizes of up to 512 bytes, a low end (by today's standards) Linux machine delivers a rate of over 70 000 messages per second, translating to just $14\mu s$ per message. Furthermore, [45] shows a great performance improvement in transfer rates for Linux IPC by using UNIX domain sockets instead of UDP. ODTONE does not support UNIX sockets yet but, given its open source nature and the message-oriented IPC mechanism, the support should be implemented with little effort.

Table 6.3 shows a comparison of connection timings between the Mobile Node and `PoA#1`, which is running a Wi-Fi network protected with WPA2 and running a DHCP version 4 (DHCPv4) server. Both connections are started from a clean state, with the Wi-Fi device powered off. The first time interval is measured from the point each solution begins the association procedure and until the link layer setup is completed (i.e., immediately before the IP configuration step). The second interval corresponds to the start of the network layer configuration procedures, and until the connection is fully completed. These steps are timed via the reception of D-Bus signals for the Device state changes. The test was repeated 50 times for *NetworkManager* and EMICOM, and the error is indicated for a confidence interval of 99%.

	NetworkManager	EMICOM
WPA Association	3.428 ± 0.020	3.353 ± 0.026
DHCP	1.233 ± 0.022	1.136 ± 0.012
Total	4.661 ± 0.028	4.490 ± 0.030

Table 6.3: Network configuration timings, in seconds.

Despite the additional overhead of the EMICOM solution, a slight improvement can be observed in both in the link and network layer setups. Several factors contribute to this result. *wpa_supplicant* is the link layer and security configuration daemon for both solutions, although *NetworkManager* uses the socket interface, and EMICOM the D-Bus mechanism. In principle, given the fact that D-Bus messages are routed via an additional daemon, this should account to worse performance from the EMICOM solution. As for DHCP, *dhclient* is used in both cases, and the interaction is done via scripted invocation by both network managers. Thus, the external factors do not contribute to the increased performance, which can only be explained via the internal routines of each solution. In fact, following the connection request by the user, before the link layer association step, *NetworkManager* spends even more time with an additional preparation phase not considered in these timings.

6.3 Code base

When compared to a native solution, the extra layer for media abstraction requires a greater amount of code for translating operations. However, the higher layers require less code for controlling individual interfaces, since the procedures are are reused across device technologies.

Table 6.4 offers a direct comparison between the provided framework and the GNU/Linux *NetworkManager* (version 0.9.6.0) solution, using the *SLOCCount*[1] tool. Despite being developed in different programming languages (C vs. C++), the number of code lines is nonetheless an acceptable measure of development effort. It should be noted that not the entire code-base of *NetworkManager* is considered; the counting excludes *NetworkManager* components not yet included in the EMICOM framework such as automatic VPN setup, Bluetooth and WIMAX support, etc.

NetworkManager	EMICOM		
Total: 70 815	MIHF (ODTONE)	11 606	Total: 21 960
	MIH User	5 277	
	802.11 Link SAP	1 610	
	802.3 Link SAP	897	
	libnl wrapper	1 347	
	dhclient wrapper	77	
	wpa_supplicant wrapper	1 146	

Table 6.4: Code base comparison, in number of source code lines.

It is clear that the whole EMICOM framework, providing the same features as the considered for the *NetworkManager* software, requires less than a third of the source code. Several reasons contribute to this fact, the most prominent being the different programming languages. Other factors include the used libraries. ODTONE and EMICOM are highly dependent on the *Boost* libraries[2] for data manipulation, which could also be a relevant contribution to the decrease in code size.

6.4 Memory usage

Process memory usage is a common limiting factor in some deployment scenarios. Embedded systems usually have limited memory. Even in desktop computers, it is desirable that resident applications account for a small impact on the overall system capacity.

Measuring process memory usage in modern OSs is a complex task. Processes commonly make use of system libraries that, once loaded into memory, can be reused several times by several processes, and thus the system does not allocate multiple instances of the library. These libraries can be considered components of a program, but the program may not be the sole responsible for loading the library into the memory.

[1] *SLOCCount* tool, `http://www.dwheeler.com/sloccount/`

[2] *Boost* Libraries, `http://www.boost.org`

The *Valgrind*[1] utilities allow developers to track memory allocations of individual processes. This utility can accurately report the memory that each process allocates both in the Heap and Stack memory segments. Table 6.5 shows the size of a snapshot of the combined Heap and Stack memory allocated by each solution, captured after an initial launch, after the attachment to both an 802.3 and 802.11 network (for this specific test, the computer is also attached by Ethernet, not represented in Figure 6.1).

NetworkManager	EMICOM		
Total: 967 064	MIHF (ODTONE)	35 688	Total: 553 496
	MIH User	401 192	
	802.11 Link SAP	52 200	
	802.3 Link SAP	64 416	

Table 6.5: Memory allocated by each solution, in bytes.

Again, comparing similar situations for both solutions, the EMICOM framework shows a great benefit, compared to the *NetworkManager* software. *NetworkManager* is openly developed, and has existed for a long time. Apart from the base ODTONE library, the EMICOM software has not been reviewed by external developers, and has not been submitted to optimization procedures, which means there could still be improvements in this area. It should be noted that the *boost* libraries do not directly contribute to this factor, since they are mostly header-only, thus not loaded as shared system libraries.

6.5 Benefits

The real world benefit for this framework is the plethora of scenarios where it may be considered for network selection and handover optimization. One of the main aspects that benefit a system with an 802.21-based networking solution is the Information Service, that will allow obtaining information about neighboring networks without powering additional radios. This service also allows the exchange of many network configurations and policies that will allow decision algorithms to take into account variables such as the cost for each network, the throughput or delay requirements for each application, and much more.

6.5.1 Battery life

This aspect is increasingly relevant, as more and more mobile devices provide multiple radios for multiple network technologies. Figure 6.2 shows two different test runs, where a single laptop computer is retrieving a video stream via a Wi-Fi interface at a fixed rate of $500KB/s$. In one test run, the laptop is running the GNU/Linux *NetworkManager*, and the second is using the EMICOM framework. Both have one Ethernet interface and two Wi-Fi interfaces. This need not be the case, as there should be little benefit in having two similar radio interfaces, but it serves to compare the impact of having more than one wireless device in the

[1] *Valgrind* utilities, `http://valgrind.org/`

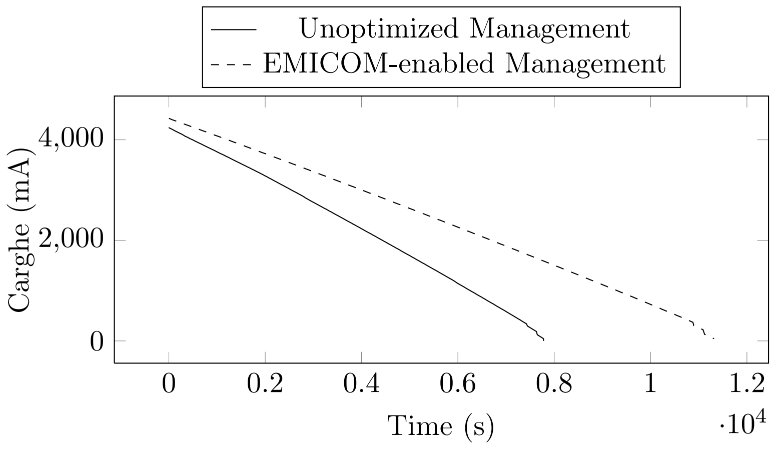


Figure 6.2: Battery drain comparison.

same system. Moreover, due to EMICOM's usage of 802.21 abstraction mechanisms, the same events and commands used in this scenario would still be valid in scenarios featuring other technologies such as 802.16 and 3GPP links.

A regular Network Manager typically employs very basic and static connectivity strategies, which implies having all the devices active at all times. In this specific scenario, we enhanced the base behaviour by allowing the second device to not be active at all times, but instead waking up at regular intervals of 30 seconds (halving in frequency every half hour) and performing a scan. A 802.21 solution, however, does not need to power the secondary device for learning about neighbouring networks, because it can rely on the Information service for the task of finding neighbouring networks, so in this scenario the MIH User always keeps the second device off, only activating it when an optimized handover opportunity occurs.

In the collected results, it is clear that the EMICOM run starts with higher initial capacity. This is common, as batteries often report different maximum capacity values at each charge cycle. The impact of powering an additional interface and performing scans is nonetheless noticeable and very significant, reducing the total autonomy of the device by a factor of 30%. Interestingly, though, the chart does not portray the increase in scan intervals after each 30 minute period. The initial interval of 30 seconds is perhaps a low initial value, resulting in maximum scan interval of 8 minutes in the experiment, although this would be an acceptable value for connectivity in a high mobility scenario.

This test does not effectively consider the real life context where network handovers would be performed, as the mobile node is always connected to the same PoA. The data exclusively highlights the potential impact on overall power consumption of the device by not having to periodically activate multiple network interfaces.

6.5.2 Optimal selection

Figure 6.3 provides a 60 second test scenario where the EMICOM tool benefits from information from an MIH User in the Network, assisting with the handover decision. In this specific test, two Wi-Fi APs offer access to the same network. The computer is trying to maximize its

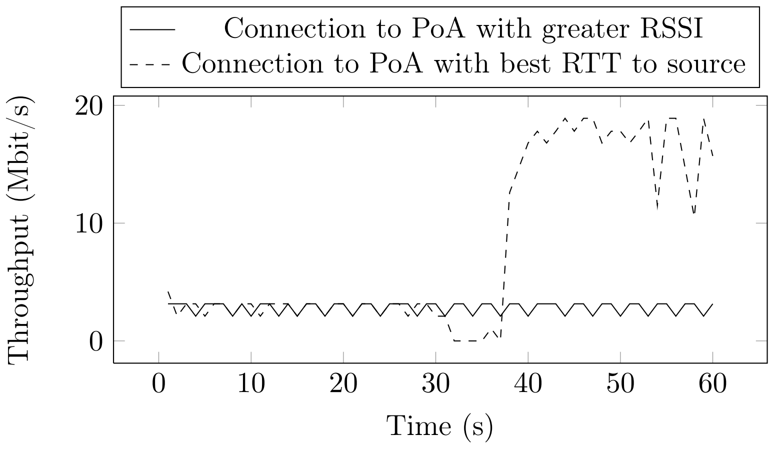


Figure 6.3: Optimal AP selection for throughput.

TCP throughput by retrieving an on-demand video from the server. One AP has a stronger signal than the other ($-23dBm$ versus $-39dBm$), but it offers a lower throughput. This could be because the stronger AP is serving a greater amount of users, or has a low downlink, or any other reasons. In the specific test case, the rate was throttled at the AP on purpose. The GNU/Linux *NetworkManager* always stays connected to the strongest AP, since it only bases its connectivity decision on signal level. EMICOM however, using 802.21, receives information pushed by the network (e.g., via a *MIH Net Handover Commit request* command) to the User after 30 seconds, suggesting a handover to the other AP. This enables better load balancing on the network, while directly benefiting the user service.

A period with total loss of connectivity is noticeable. This is the occurrence of the hard handover, since the integration with AP mobility management engines achieving seamless mobility are out of scope from this dissertation. However, EMICOM, is able to leverage from mobility management primitives provided by 802.21, when such IP mobility schemes are employed.

Nonetheless, via the stimuli provided by the 802.21-enabled Network Manager, the EMICOM framework was able to enhance the throughput of the video reception by performing a link switch to an AP with better downlink connectivity.

Chapter 7

Conclusions and Future Work

Network Manager applications are becoming standard in different Operating Systems, but lack the flexible capabilities for abstract access technology interfacing as well as disseminating and receiving handover optimization information from other sources, both local to the node or remotely available in network controlling entities.

The proposed framework, more than just integrating 802.21 with Network Manager functionality, extended the first with the support for security association and address negotiation procedures, which are invaluable in today's connectivity procedures in mobile networks. Apart from the proposed extensions to the standard, this project also produced standard Link SAP implementations on top of current hardware and Linux kernel facilities, to the extent possible, in integration with the ODTONE project.

The implementation of EMICOM also allowed the realization of an extensive evaluation effort, providing insight on the benefits of using Media Independent information and control capabilities to assist optimized handover and interface selection. The obtained results, in seamlessly replacing the popular *NetworkManager* from the GNU/Linux Operating System, showed a reduced code base, better battery consumption and support for opportunistic network attachment and optimized handover procedures.

Personally, this work enriched the author's skills with a greater understanding of network management procedures, as well as the specific case of GNU/Linux network programming. The privilege of working with state of the art standards for future networks also constituted an important chance for developing autonomous and effective work methods.

7.1 Contributions

Network management in mobile networks is currently an important topic of discussion in the telecommunications fora. This work presents yet another contribution to this discussion, by providing a proof of concept architecture and preliminary results for ubiquitous network management. The obtained results were presented at *Conferência sobre Redes de Computadores*[1] in November of 2012.

[1] *Conferência sobre Redes de Computadores*, `http://crc2012.av.it.pt/`

The *MultiMEDia transport for mobilE Video AppLications* (MEDIEVAL)[1] project of the FP7 program, by the European Commission, is researching the architecture and inherent mechanisms for evolving today's Internet towards the efficient support of video services over wireless access networks with mobility support, commercially deployable by operators. It relies on the IEEE 802.21 standard, also via ODTONE, as an enabler of interactions between the different access technologies and high-level decision modules. This dissertation, developed in the same research group as the ODTONE framework, contributed to a broad test of the library's features and capabilities, as well as source code fixes and suggestions to the project; this included direct support given to different partners of the project's consortium, belonging to different international research institutions and industry. Also, the practical use of the library allowed for a vast contribution to other users in the form of documentation and assistance in the mailing lists, regarding a variety of issues and technical difficulties. The results were also included in a technical report

Furthermore, the developed 802.11 Link SAP was adopted by the MEDIEVAL project, increasing the derived project with media independent control to Wi-Fi interfaces, as well as providing a complex testing environment for the work presented here. Finally, it was also deployed on the demonstration testbed which showcased several demonstrations for the project's second year audit, where an evaluation mark of EXCELENT was given. A project's technical report includes the description of the developed Link SAP, as well as the results from this dissertation.

Portions of the software were also included in an Alcatel-Lucent Bell Labs[2] open day demonstration for seamless connectivity handover between Wi-Fi and cellular technologies.

The software will be publicly available at `https://github.com/ATNoG/EMICOM`.

7.2 Future work

In its current state, the EMICOM framework does not fulfill the OS-independence requisite for Network Managers. This can only be achieved by the reimplementation of the Link SAP functionality for different platforms. Also, at the moment, the MIH User is tightly coupled with the D-Bus IPC framework which, although ported and tested in various Operating Systems, is mainly focused on GNU/Linux.

The current NetworkManager interface is not equipped with appropriate mechanisms for gathering application requirements and user preferences for network selection. First, the D-Bus interface needs to be expanded with the methods for applications to request certain parameters, and for the users to be able to specify advanced requirements such as network cost per duration or per bit. This then has to be put together in the decision algorithms for network selection, following some prioritization criteria.

Support for mobility protocols should also be integrated in the network setup procedures, for seamless handover capabilities. The work presented in [6] is a success case for a network-

[1]MEDIEVAL, `http://www.ict-medieval.eu/`

[2]Alcatel-Lucent Bell Labs, `http://www.alcatel-lucent.com/wps/portal/belllabs`

controlled mobility scenario using 802.21, specifically the ODTONE implementation.

Finally, the 802.21 architecture depends on the configuration of MIH nodes to discover each other. The dynamic procedures proposed for this discovery, based on DHCP and DNS, have been tested in [57], using ODTONE as the base framework as well.

Bibliography

[1] “The World in 2011: ICT Facts and Figures,” 2012. 1

[2] Abid, M. and Yahiya, T.A. and Pujolle, G., “A Utility-based Handover Decision Scheme for Heterogeneous Wireless Networks,” in *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, vol. , pp. 650 –654, jan. 2012. 1.1

[3] Gustafsson, E. and Jonsson, A., “Always best connected,” *Wireless Communications, IEEE*, vol. 10, pp. 49 – 55, feb. 2003. 1.1

[4] Kassar, M. and Achour, A. and Kervella, B., “A mobile-controlled handover management scheme in a loosely-coupled 3G-WLAN interworking architecture,” in *Wireless Days, 2008. WD ’08. 1st IFIP*, vol. , pp. 1 –5, nov. 2008. 1.1

[5] Bertin, P. and Guillouard, K. and Rault, J.-C., “IP based network controlled handover management in WLAN access networks,” in *Communications, 2004 IEEE International Conference on*, vol. 7, pp. 3906 – 3910 Vol.7, june 2004. 1.1

[6] Corujo, D. and Guimaraes, C. and Santos, B. and Aguiar, R.L., “Using an open-source IEEE 802.21 implementation for network-based localized mobility management,” *Communications Magazine, IEEE*, vol. 49, pp. 114 –123, September 2011. 1.1, 7.2

[7] “IEEE Standard for Local and Metropolitan Area Networks- Part 21: Media Independent Handover,” *IEEE Std 802.21-2008*, pp. c1 –301, 21 2009. 1.1

[8] R. Braden, “Requirements for Internet Hosts - Communication Layers.” RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633. 2

[9] “IEEE Standard for Local and Metropolitan Area Networks Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Section One,” *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pp. c1 –597, 26 2008. 2.1.1

[10] “IEEE Standard for Local and Metropolitan Area Networks Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, pp. 1 –1076, 12 2007. 2.1.2, 5.4.2

[11] Eklund, C. and Marks, R.B. and Stanwood, K.L. and Wang, S., "IEEE standard 802.16: a technical overview of the WirelessMAN air interface for broadband wireless access," *Communications Magazine, IEEE*, vol. 40, pp. 98 –107, june 2002. 2.1.3

[12] 3GPP, "General Packet Radio Service (GPRS); Service description; Stage 2," TS 03.60, 3rd Generation Partnership Project (3GPP), Oct. 2002. 2.1.4

[13] 3GPP, "UMTS Phase 1," TS 22.100, 3rd Generation Partnership Project (3GPP), Oct. 2001. 2.1.4

[14] 3GPP, "3GPP system architecture evolution (SAE): Report on technical options and conclusions," TR 23.882, 3rd Generation Partnership Project (3GPP), Sept. 2008. 2.1.4

[15] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access (E-UTRAN); Overall description; Stage 2," TS 36.300, 3rd Generation Partnership Project (3GPP), Sept. 2008. 2.1.4

[16] 3GPP, "Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface," TS 11.11, 3rd Generation Partnership Project (3GPP), June 2007. 2.2

[17] "IEEE Standard for Local and Metropolitan Area Networks Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-1997*, pp. i –445, 1997. 2.2

[18] Mantin, Itsik, "A practical attack on the fixed RC4 in the WEP mode," in *Proceedings of the 11th international conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'05, (Berlin, Heidelberg), pp. 395–411, Springer-Verlag, 2005. 2.2

[19] Rafik Chaabouni, "Break WEP Faster with Statistical Analysis," tech. rep., EPFL, LASEC, June 2006. 2.2

[20] Tews, Erik and Weinmann, Ralf-Philipp and Pyshkin, Andrei, "Breaking 104 Bit WEP in less than 60 seconds," in *Proceedings of the 8th international conference on Information security applications*, WISA'07, (Berlin, Heidelberg), pp. 188–202, Springer-Verlag, 2007. 2.2

[21] "IEEE Standard for Local and Metropolitan Area Networks Part 11, Amendment 6: Medium Access Control (MAC) Security Enhancements," *IEEE Std 802.11i-2004*, pp. 1 –175, 2004. 2.2

[22] "IEEE Standard for Local and metropolitan area networks - Port-Based Network Access Control," *IEEE Std 802.1X-2010 (Revision of IEEE Std 802.1X-2004)*, pp. C1 –205, 5 2010. 2.2

[23] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz, "Extensible Authentication Protocol (EAP)." RFC 3748 (Proposed Standard), June 2004. Updated by RFC 5247. 2.2

[24] J. Postel, "Internet Protocol." RFC 791 (Standard), Sept. 1981. Updated by RFCs 1349, 2474. 2.3

[25] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564. 2.3

[26] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)." RFC 3022 (Informational), Jan. 2001. 2.3

[27] J. Postel, "Internet Control Message Protocol." RFC 792 (Standard), Sept. 1981. Updated by RFCs 950, 4884, 6633. 2.3

[28] A. Conta, S. Deering, and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification." RFC 4443 (Draft Standard), Mar. 2006. Updated by RFC 4884. 2.3

[29] R. Droms, "Dynamic Host Configuration Protocol." RFC 2131 (Draft Standard), Mar. 1997. Updated by RFCs 3396, 4361, 5494. 2.3

[30] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)." RFC 3315 (Proposed Standard), July 2003. Updated by RFCs 4361, 5494, 6221, 6422, 6644. 2.3

[31] S. Thomson, T. Narten, and T. Jinmei, "IPv6 Stateless Address Autoconfiguration." RFC 4862 (Draft Standard), Sept. 2007. 2.3

[32] P. Mockapetris, "Domain names - concepts and facilities." RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. 2.3

[33] P. Mockapetris, "Domain names - implementation and specification." RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604. 2.3

[34] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, August 2007. 3

[35] R. Love, *Linux Kernel Development.* Addison-Wesley Professional, 3rd ed., 2010. 3.1.1, 3.1.2

[36] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005. 3.1.1

[37] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 1st ed., 2010. 3.1.3, 3.2

[38] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov, "Linux Netlink as an IP Services Protocol." RFC 3549 (Informational), July 2003. 3.1.3

[39] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in linux using netlink sockets," *Softw. Pract. Exper.*, vol. 40, pp. 797–810, Aug. 2010. 3.1.3

[40] N. Horman, "Understanding and programming with netlink sockets." `http://people.redhat.com/nhorman/papers/netlink.pdf`. Online; accessed September 2012. 3.1.3.1

[41] "nl80211 documentation - linux wireless." `http://wireless.kernel.org/en/developers/Documentation/nl80211`. Online; accessed September 2012. 3.1.3.2

[42] "Magic Packet Technology," tech. rep., Advanced Micro Devices (AMD), 1995. 3.1.3.2

[43] "Regulatory documentation - linux wireless." `http://wireless.kernel.org/en/developers/Regulatory`. Online; accessed September 2012. 3.1.3.2

[44] "IEEE Standard for Local and metropolitan area networks Part 11, Amendment 2: Fast Basic Service Set (BSS) Transition," *IEEE Std 802.11r-2008 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008)*, pp. 1 –126, 15 2008. 3.1.3.2

[45] Wright, K. Gopalan, and H. Kang, "Performance analysis of various mechanisms for inter-process communication," 2007. 3.2.1, 6.2

[46] Artemio, S. and Leonardo, B. and Hugo, J. and Carlos, M.J. and Aceves-Fernandez, M.A. and Carlos, P.J., "Evaluation of CORBA and Web Services in distributed applications," in *Electrical Communications and Computers (CONIELECOMP), 2012 22nd International Conference on*, pp. 97 –100, Feb 2012. 3.2.2

[47] Shang-Fu, Gong and Xiao-Li, Yang, "Study and Design of Integrated Transmission Network Management System Based on CORBA and Web," in *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pp. 600 –603, Aug. 2012. 3.2.2

[48] Ferenc, G. and Dimic, Z. and Lutovac, M. and Vidakovic, J. and Kvrgic, V., "Distributed robot control system implemented on the client and server PCs based on the CORBA protocol," in *Embedded Computing (MECO), 2012 Mediterranean Conference on*, pp. 158 –161, June 2012. 3.2.2

[49] Zhang Haibo and Li Yang, "Research and development of distributed data integration query system based on CORBA," in *Innovative Smart Grid Technologies - Asia (ISGT Asia), 2012 IEEE*, pp. 1 –4, May 2012. 3.2.2

[50] Huang, Min and Zhu, Lizhe, "Research for Network Fault Real-time Alarm System Based on Pushlet," in *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pp. 212 –215, Aug. 2012. 3.2.2

[51] Henning, Michi, "The Rise and Fall of CORBA," *Queue*, vol. 4, pp. 28–34, Jun 2006. 3.2.2

[52] K. Vervloesem, "Control your linux desktop with d-bus," *Linux J.*, vol. 2010, Nov. 2010. 3.2.3.4

[53] de la Oliva, A. and Bernardos, C.J. and Calderon, M. and Melia, T. and Zuniga, J.C., "IP flow mobility: smart traffic offload for future wireless networks," *Communications Magazine, IEEE*, vol. 49, pp. 124 –132, oct. 2011. 4.1

[54] "IEEE Draft Standard for Local and metropolitan area networks Part 11, Amendment 7: Interworking with External Networks," *IEEE Unapproved Draft Std P802.11u/D5.0, Feb 2009*, 2009. 4.2.4, 5.4.2

[55] K. McCloghrie and F. Kastenholz, "The Interfaces Group MIB." RFC 2863 (Draft Standard), June 2000. 5.4.3

[56] B. F. G. Bidulock, "Streams vs. sockets performance comparison for udp," *OpenSS7*, 2007. 6.2

[57] D. Corujo, C. Guimaraes, and R. Aguiar, "Evaluation of Discovery Mechanisms for Media Independent Handover Services," in *Proc. IEEE International Conference on Communication 2012 Workshop on Convergence among Heterogeneous Wireless Systems in Future Internet*, June 2011. 7.2

[58] B. Stroustrup, *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. A

Appendix A

Memory Management

The *boost* libraries offer macros that facilitate memory management based on scope context. The `BOOST_SCOPE_EXIT` macro, for example, can be used to define code that is to be executed when leaving the scope where it is defined, either by common returns or due to exception throwing. For example:

```
void function() {
    struct rtnl_addr *addr = rtnl_addr_alloc();
    BOOST_SCOPE_EXIT( (addr) ) {
        if (addr) rtnl_addr_put(addr);
    } BOOST_SCOPE_EXIT_END

    if (!rand())
        throw "small chance exception"; // addr is deleted

    return; // addr is deleted
}
```

Although the macro is very useful, the memory management code is still visible, and the code defined in the macro itself cannot fail, or else it will cause memory leaks as well. Bjarne Stroustrup invented a concept to avoid resource leaks in C++ programs, called RAII[58]. This technique consists of creating wrappers for leveraging the automatic memory management of C++ for resource management in this manner:

```
class Addr {                                   |
public:                                        | void function() {
  Addr() {                                     |   Addr a;
    addr = rtnl_addr_alloc();                  |   return;
    if (!addr) throw "memory exception";       |   // a is destroyed
  }                                            | }
  ~Addr() {                                    |
    rtnl_addr_put(addr);                       |
  }                                            |
private:                                       |
  rtnl_addr *addr;                             |
}                                                 |
```

This method provides a better solution than the previous one because the resulting code is much cleaner, without explicit memory management procedures. Moreover, the memory management code is concealed only once, for multiple usages, whilst scope macros would have to be used for every scope.

It should be noted that a RAII wrapper also requires some caution regarding object creation. If a class constructor does not terminate normally, the object is not constructed and the destructor is not invoked. If the object inherits from another class, the destructors of the base classes are invoked, though, because the base objects are always constructed first.

Appendix B

EMICOM Command Line Parameters

The following program listings show the command line parameters available to each EMICOM component. Starting with the options for the MIHF, followed by the Link SAPs, and finally the NetworkManager MIH User. In the case of the Link SAP options, all are available across both the 802.11 and 802.3 Link SAPs, except for the "Scheduled scan interval" option, which refers only to 802.11 options.

```
MIHF Configuration Options:
  --help                                   Display configuration options
  --conf.file arg (=odtone.conf)           Configuration file
  --conf.recv_buff_len arg (=4096)         Receive buffer length
  --mihf.id arg (=mihf)                    MIHF ID
  --mihf.ip arg (=127.0.0.1)               MIHF IP
  --mihf.remote_port arg (=4551)           Remote MIHF communication port
  --mihf.local_port arg (=1025)            Local SAPs communications port
  --mihf.peers arg                         List of peer MIHFs
  --mihf.users arg                         List of local MIH-Users
  --mihf.links arg                         List of local Links SAPs
  --mihf.transport arg (=udp)              List of supported transport protocols
  --mihf.link_response_time arg (=3000)    Link SAP response time (milliseconds)
  --mihf.link_conf_response_time arg (=10000) Link Conf response time (milliseconds)
  --mihf.l3_conf_response_time arg (=10000)   L3 Conf response time (milliseconds)
  --mihf.link_delete arg (=2)              Link SAP response fails to forget
  --mihf.discover arg                      MIHF Discovery Mechanisms Order
  --enable_multicast                       Allows multicast messages
  --enable_unsolicited                     Allows unsolicited discovery
  --log arg (=1)                           Log level [0-4]

MIH Link SAP Configuration:
  --help                                      Display configuration options
  --link.verbosity arg (=2)                   Log level [0-2]
  --link.sched_scan_period arg (=0)           Scheduled scan interval (millis)
  --link.default_th_period arg (=1000)        Threshold check interval (millis)
  --link.link_addr arg                        Interface address
  --link.port arg (=1235)                     Port
  --conf.file arg (=sap_80211.conf)           Configuration File
  --conf.recv_buff_len arg (=4096)            Receive Buffer Length
  --mihf.ip arg (=127.0.0.1)                  Local MIHF Ip
  --mihf.local_port arg (=1025)               MIHF Local Communications Port
  --mihf.id arg (=local-mihf)                 Local MIHF Id
  --link.id arg (=link)                       Link SAP Id
  --sys.resolv_conf_file arg (=/etc/resolv.conf) System's resolv.conf location
```

```
MIH Usr Configuration:
  --help                                Display configuration options
  --conf.file arg (=networkmanager.conf) Configuration file
  --conf.recv_buff_len arg (=4096)      Receive buffer length
  --conf.port arg (=1234)               Listening port
  --user.id arg (=mih_nm)               MIH-User ID
  --mihf.ip arg (=127.0.0.1)            Local MIHF IP address
  --mihf.local_port arg (=1025)         Local MIHF communication port
  --dest arg                            MIHF destination
  --nm.settings_path arg (=./settings)  Path for NetworkManager settings persistence
  --nm.version arg (=0.9.6.0)           NetworkManager version to mimic
  --nm.networking_enabled arg (=1)      NetworkingEnabled property initial value
  --nm.wireless_enabled arg (=1)        WirelessEnabled property initial value
  --nm.wimax_enabled arg (=1)           WirelessEnabled property initial value
  --nm.wwan_enabled arg (=1)            WirelessEnabled property initial value
```