Wireless Extensions for Linux

Jean Tourrilhes

23 January 97

A Wireless LAN API for the Linux operating system.

1 Introduction

The purpose of this document is to give an overview of the **Wireless Extensions**. The first part will explain the reason of this development and the generals ideas behind this work. Then, we will give the current status of the development. The third part will explore the user interface. We will finish by some implementation details and the possible evolution of the Wireless Extensions.

This document is a part of the Linux Wireless LAN Howto. Please refer to it for details.

2 Philosophy & Goal

It all started when I tried to install a **Wavelan** network on **Linux** computers. I was having a ISA and a PCMCIA versions of the Wavelan, and the two drivers were using totally different methods for the setup and collection of statistics (and in fact fairly incomplete...). As my small hard disks didn’t allow me to reboot to DOS to set up those missing parameters, I started to modify the driver code.

I decided to define a wireless API which would allow the user to manipulate any wireless networking device in a **standard** and **uniform** way. Of course, those devices are fundamentally different, so the standardisation would only be on the methods but not on the values (a Network ID is always a parameter that you may set and get and use to distinguish logical networks, but some devices might use 4 bits and others 16 bits, and the effect of a change may be immediate or delayed after a reconfiguration of the device...).

This interface would need to be flexible and extensible. The primary need was for device **configuration**, but wireless **statistics** and the development of **wireless aware applications** was desirable. I needed also something simple to implement and conform to the Linux standard to have something much easier to share and maintain. The interface would need to evolve with the apparition of new products and with specific needs.

I tried to be as generic as possible, but I was obliged to restrict myself on a specific set of devices. I focused on the **Wireless LAN** type of devices (indoor radio networks), which appears in the operating system as a normal network device. **AX25** (Amateur Radio) devices are too specific and have already their own specific network stack and tools. I also ruled out **cellular data** (data over GSM and mobile phone) and medium and long distance wireless communications because their interface are quite different and they don’t operate in the same way. **InfraRed** should be similar enough to Radio LANs to benefit from this interface (if someday an InfraRed network driver for Linux appears).

As it was only an extension to the current Linux networking interface, I decided to call it “Wireless Extensions”. The words interface and API are too ambitious for this simple set of tools.

3 Availability

The wireless extensions have been implemented in three complementary parts. The first part is the user interface, a set of tools to manipulate those extensions. The second part is a modification of the Linux kernel to support and define the extensions. The third part is the hardware interface and is implemented in each network driver itself to map the extensions to the actual hardware manipulations.

The kernel modifications have been included in the last Linux releases (from 2.0.30 and 2.1.17) and so the kernel now supports them. Users of release 1.2.13 may use a patch to add the support in their kernels. By default, the wireless extensions are disabled, and users need to enable the *CONFIG_NET_RADIO* option in the kernel configuration (this is the option which enables the choice of radio interfaces in the network driver list).

The tools are quite simple and so should be able to compile in any Linux system provided that the kernel supports wireless extensions.

The modifications of the wireless network drivers are probably the most important. Each driver needs to support wireless extensions and to perform the corresponding dialogue with the specific hardware. For now, only the Wavelan ISA and the Wavelan PCMCIA drivers support wireless extensions, but I’m confident that others will eventually follow. The modified Wavelan PCMCIA driver is available in the last pcmcia packages (from 2.9.1). The modified Wavelan ISA driver is available in the kernel (from 2.0.30 and 2.1.17).

4 User interface and tools

The user interface is composed of 3 programs and a /proc entry. The main goal of the Wireless Extension development effort wasn’t the user interface, so this interface is quite basic.

4.1 /proc/net/wireless

This is a /proc entry. /proc is a pseudo file system giving information and statistics about the current system, which is usually located in /proc. These entries act as files, so a **cat** on them will give the required information. /proc/net/wireless is designed to give some wireless specific statistics on each wireless interface in the system. This entry is in fact a clone of /proc/net/dev which gives the standard driver statistics.

The output looks like this :

```
<jean@tourrilhes>cat /proc/net/wireless
Inter-|sta|  Quality         |  Discarded packets
```

```
face |tus|link level noise| nwid crypt  misc
 eth2: f0   15.  24.    4.   181     0     0
```

For each device, the following information is given :

- *Status* : Its current state. This is a device dependent information.
- *Quality - link* : general quality of the reception.
- *Quality - level* : signal strength at the receiver.
- *Quality - noise* : silence level (no packet) at the receiver.
- *Discarded - nwid* : number of discarded packets due to invalid network id.
- *Discarded - crypt* : number of packet unable to decrypt.
- *Discarded - misc* : unused (for now).

These informations allow the user to have a better feedback about his system. A high value of *Discarded - nwid* packet might indicate a nwid configuration problem or an adjacent network. The *Quality - level* might help him to track shadow areas.

The basic difference between *Quality - link* and *Quality - level* is that the first indicate how good the reception is (for example the percentage of correctly received packets) and the second how strong the signal is. The *Quality - level* is some directly measurable data that is likely to have the same signification across devices.

When the *Quality* values have been updated since the last read of the entry, a dot will follow that value (typically, it mean that a new measure has been made).

4.2 iwconfig

This tool is designed to configure all the wireless specific parameters of the driver and the hardware. This is a clone of **ifconfig** used for standard device configuration. The following parameters are available :

- *freq* or *channel* : the frequency or the channel sequence.
- *nwid* : network id or domain, to distinguish different logical networks.
- the *name* of the "protocol" used on the air.
- *sens* : this is the signal level threshold to trigger packet reception (sensitivity).
- *enc* : the encryption or scrambling key used.

The frequency or channel parameter is the physical separation between networks (the keyword *freq* and *channel* are synonymous). For frequency hopping devices, it might be the hopping pattern. On the other hand, the nwid is only a logical (virtual) separation between networks which might be on the same frequency.

The name of the protocol is often the generic name of the device itself (for example *"Wavelan"*). This is quite useful because these protocols are all incompatible. The apparition of standards such as **802.11** and **HiperLan** might help a bit in the future.

The encryption setting includes the key itself (up to 64 bits), the activation of encryption (*on/off*) and an optional argument (either the type of algorithm or the key number for multi key systems).

Without any argument, **iwconfig** gives the value of these parameters (and the content of /proc/net/wireless) :

```
<jean@tourrilhes>iwconfig
lo        no wireless extensions.

eth1      no wireless extensions.

eth2      Wavelan  NWID:1234  Frequency:2.422GHz  Sensitivity:4
          Link quality:15/15  Signal level:22/63  Noise level:0/63
          Rx invalid nwid:181  invalid crypt:0  invalid misc:0
```

By giving a command line option, the user may change these parameters (if it is possible in the driver and if the user is root). For example, to change the frequency to *2.462 GHz* and disable nwid checking on the device *eth2* , the user will do :

```
#iwconfig eth2 freq 2.462G nwid off
```

The user may also list the available frequencies for a specific device or the number of channels defined :

```
<jean@tourrilhes>iwconfig eth2 list_freq
10 channels ; available frequencies : 2.422GHz  2.425GHz  2.4305GHz  2.432GHz
2.442GHz  2.452GHz  2.46GHz  2.462GHz
```

4.3 iwspy

iwspy was designed to test the **Mobile IP** support. It allows the user to set a list of network addresses in the driver. The driver will gather quality information for each of those addresses (updated each time it receives a packet from that address). The tool allows the user to display the information associated with each address in the list.

iwspy accept IP address as well as hardware addresses (MAC). IP addresses will be converted to hardware addresses before being transmitted to the driver. No verification is made on the hardware address. On the command line, hardware addresses should be prefixed by the keyword *hw* :

```
#iwspy eth2 15.144.104.4 hw 08:00:0E:21:3A:1F
```

The tool accept the keyword “+”to add the new addresses at the end of the list :

```
#iwspy eth2 + hw 08:00:0E:2A:26:FA
```

To display the list of address :

```
<jean@tourrilhes>iwspy eth2
08:00:0E:21:D7:4E : Quality 15 ; Signal 29 ; Noise 0 (updated)
08:00:0E:21:3A:1F : Quality 0 ; Signal 0 ; Noise 0
08:00:0E:2A:26:FA : Quality 0 ; Signal 0 ; Noise 0
```

The *(updated)* indication show that we have received a packet since we set the address in the driver. We haven't received anything from the second and third address (this explain why the values are 0). A **ping** to those address should solve the problem.

The number of addresses is limited to 8, because each address slow down the driver (it increases the processing for each received packets).

4.4 iwpriv

iwpriv is some experimental support for device specific extensions. Some drivers (like the Wavelan one) might define some extra parameters or functionality, **iwpriv** is used to manipulate those.

5 Driver interface & programming issues

The implementation of the wireless extensions was designed to minimise the number of changes in the kernel and to have a simple and extensible solution.

All the wireless interface (types and constants) is defined in the file /usr/include/linux/wireless.h. This is the central piece of the wireless extension.

5.1 /proc/net/wireless

This /proc entry is a clone of the /proc/net/dev entry and has been implemented in exactly the same way.

The Linux networking stack uses a structure (*struct device*) to keep track of each device in the system. The first part of it is standardised, and contains parameters (for example the base I/O address and the IP address of the device) and callbacks (the procedure to start the device, to send a packet...).

I've added to this structure another standard callback (*get_wireless_stats*) to get the wireless statistics that /proc/net/wireless needs. When the /proc entry is read, it calls this callback for all the devices present in the system and display the information it gets. If a device doesn't define this callback, it is ignored.

When called, the *get_wireless_stats* callback returns a structure (*struct iw_statistics*) containing all the fields that will be displayed by the /proc entry. This structure is of course defined in the file /usr/include/linux/wireless.h.

5.2 ioctl

The usual method in Unix to set and get parameters from a network device is through **ioctl**. Ioctl are usually operations performed on a file descriptor, but they also apply on network sockets. The ioctl is a kernel system call. The arguments of the ioctl define the operations to be done, the parameters of these operations and the device they applies to.

For example, to change the IP address on device *eth2*, a program (like **ifconfig**) would make an ioctl call with the following parameters: *"eth2", SIOCSIFADDR, new address*. The structure defining the parameters layout may be found in /usr/include/linux/if.h and the ioctl call (the constant *SIOCSIFADDR*) is defined in /usr/include/linux/sockios.h.

For the wireless extensions, I've defined a new set of ioctl calls (for example *SIOCSIWFREQ* to change the frequency). I've also defined the parameters to those calls (see /usr/include/linux/wireless.h). In fact those new calls map closely to the functionality offered by the wireless tools.

There is another popular way to set parameters in a network driver in Linux, which is initialisation parameters (either passed on the kernel command line or at module initialisation). This method has the disadvantage that the parameters may only be set (and not read) and only at initialisation time, which offers much less flexibility than the current solution. The other advantage of ioctl is that it is a programming interface (and not only a user interface), so any program (such as a Mobile IP implementation) may manipulate them directly.

5.3 More details

For people who needs more details on the actual implementation, the list of ioctl calls or the parameters definitions, the ultimate reference is the source code. The file /usr/include/linux/wireless.h defines all the necessary pieces. The wireless tools offer some good examples on how to use the ioctl calls. To know how the wireless extensions are used on the driver side, the Wavelan driver source will answer most questions.

6 Areas of improvement

6.1 Other drivers

For now, only the Wavelan driver supports the wireless extensions. The implementation of the Wireless Extensions in other wireless network drivers might help to validate the Wireless Extensions genericity and to refine its definition. The problem is that quite few wireless devices have Linux drivers available.

Also the implementation of wireless extensions in some case is not that easy. For example, the code to change the frequency in the Wavelan driver is quite large and complex.

6.2 Users tools

They should be extended and refined. We could also imagine the development of a graphical interface (for example to have a small graph bar like for battery charge and system load).

6.3 Wireless Aware applications

The development of wireless aware applications will allow to demonstrate the concept and to expand the wireless extensions with the needs of those applications. **Mobile IP** seems an obvious target.