

# Driver réseau

**Christophe BLAESS**

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>



Ingénierie et formations sur Linux et les logiciels libres  
<https://www.logilin.fr>

<b>Interfaces réseau.....</b>	<b>3</b>
Interfaces et protocoles réseau.....	3
Sous-système net_device.....	6
<b>Driver réseau.....</b>	<b>7</b>
Enregistrement.....	7
Activation d'interface.....	8
Travaux pratiques : enregistrement d'un driver netdevice.....	9
Socket buffers.....	11
Périphériques virtuels liés.....	14
Travaux pratiques : driver d'interfaces « miroirs ».....	15
Statistiques d'utilisation.....	16
Travaux pratiques : statistiques d'utilisation d'une interface réseau.....	17

Ce support de formation est distribué sous licence **Creative Commons 4.0**



*(Attribution - Partage dans les mêmes conditions).*

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

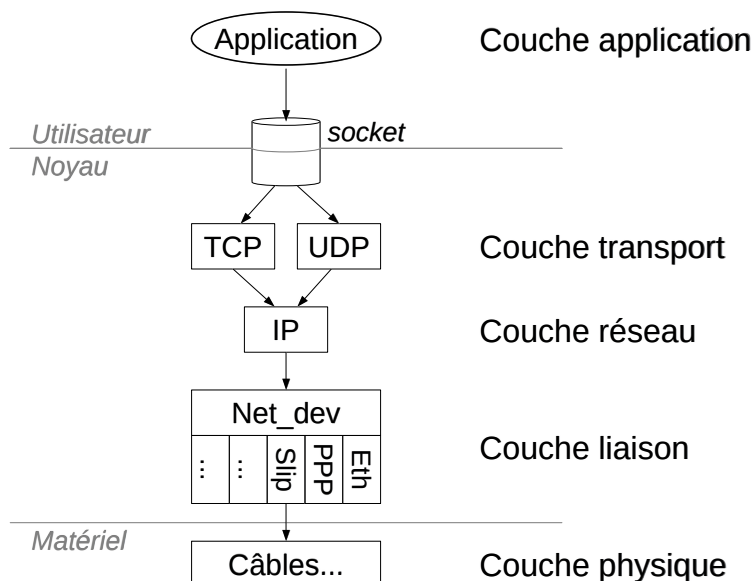
Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

v. 9.12

# Interfaces réseau

## Interfaces et protocoles réseau



La différence entre le modèle OSI classique et le modèle simplifié TCP/IP est l'absence dans ce dernier des couches *session* et *présentation* que l'on fusionne dans la couche *application*.

La pile des protocoles Internet utilisée sous Linux repose sur cinq couches, la plus élevée étant traitée par l'espace utilisateur, qui implémente les protocoles comme *HTTP*, *FTP*, *telnet*, etc. en s'appuyant sur des *sockets* – descripteurs fournis par le noyau.

Les couches de *transport* et de *réseau* sont implémentées dans le noyau indépendamment du support matériel.

La couche de *liaison* regroupe des *drivers* capable d'assurer la communication avec le matériel.

### Voir aussi

[BENVENUTI 2005] : Christian Benvenuti – *Understanding Linux Network Internals* – O'Reilly, 2005.

Les interfaces disponibles sont visibles dans `/sys/class/net` :

```
$ ls /sys/class/net/  
eth0  eth1  lo
```

ou avec la commande `ip link [show]` qui donne aussi l'adresse MAC :

```
$ ip link
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode  
DEFAULT group default qlen 1  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
  
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast  
state DOWN mode DEFAULT group default qlen 1000  
    link/ether b0:48:7a:83:3f:88 brd ff:ff:ff:ff:ff:ff  
  
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state  
UP mode DEFAULT group default qlen 1000  
    link/ether 00:1d:09:23:b5:8d brd ff:ff:ff:ff:ff:ff
```

La commande classique `ifconfig` est considérée comme obsolète et remplacée par `ip`, plus polyvalente.

Les adresses IP sont visibles en appelant `ip addr [show]` :

```
$ ip addr show
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever

2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast
state DOWN group default qlen 1000
    link/ether b0:48:7a:83:3f:88 brd ff:ff:ff:ff:ff:ff

3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 00:1d:09:23:b5:8d brd ff:ff:ff:ff:ff:ff
    inet 192.168.3.1/24 brd 192.168.3.255 scope global dynamic eth1
        valid_lft 37709sec preferred_lft 37709sec
    inet6 fe80::21d:9ff:fe23:b58d/64 scope link
        valid_lft forever preferred_lft forever
```

La table de routage avec `ip route [show]` :

```
$ ip route
```

```
default via 192.168.3.254 dev eth1 proto static metric 100
169.254.0.0/16 dev virbr0 scope link metric 1000 linkdown
192.168.3.0/24 dev eth1 proto kernel scope link src 192.168.3.1 metric 100
```

La commande `route` est également obsolète et remplacée par `ip`.

## Sous-système net\_device

```
<linux/netdevice.h>

struct net_device {
    char          name[IFNAMSIZ];
    unsigned long  features;
    unsigned int   flags;
    unsigned       mtu;
    unsigned short type;
    unsigned short hard_header_len;
    unsigned char  dev_addr[MAX_ADDR_LEN];
    unsigned char  broadcast[MAX_ADDR_LEN];
    void           *ml_priv;
    struct net_device_stats stats;
    [...]
    struct net_device_ops *netdev_ops;
    struct header_ops      *header_ops;
    [...]
}

struct net_device_ops {
    int (*ndo_open) (struct net_device *dev);
    int (*ndo_stop) (struct net_device *dev);
    int (*ndo_start_xmit) (struct sk_buff *skb,
                          struct net_device *dev);
    [...]
}
```

La structure `net_device` représente intégralement un périphérique réseau enregistré dans le noyau. Elle comprend des champs remplis par le driver et d'autres gérés par le noyau.

Les méthodes enregistrées dans la structure `net_device_ops` décrivent le comportement du périphérique, notamment dans les cas suivants :

- `ndo_open()` : affectation d'une adresse et activation par `ifconfig` ;
- `ndo_stop()` : désactivation par `ifconfig` ;
- `ndo_start_xmit()` : demande (par la couche de protocoles IP) d'émission d'un paquet.

Il faudra également fournir une méthode `create()` dans la structure `header_ops` pour remplir l'en-tête MAC des paquets sortants.

# Driver réseau

## Enregistrement

```
<linux/netdevice.h>

struct net_device * alloc_netdev (int sizeof_priv,
                                   const char *name,
                                   unsigned char name_assign_type,
                                   void (* setup)(struct net_device *));

void free_netdev (struct net_device *dev);
```

La zone de données privées (de *sizeof\_priv*) octets est allouée durant l'appel de `alloc_netdev()`. Elle est située juste après la structure `net_device` et est alignée sur une frontière de 32 bits. On y accède avec :

```
void * netdev_priv (struct net_device *dev);
```

La fonction `free_netdev()` libère aussi la zone de données privées.

Une fois la structure `net_dev` allouée et correctement remplie, on l'enregistre avec :

```
int register_netdevice (struct net_device *dev);
int unregister_netdevice (struct net_device *dev);
```

La chaîne de caractères `name` transmise à `alloc_netdev()` contient un indicateur `%d` qui sera remplacé par le numéro de l'interface de ce type.

L'argument `name_assign_type` de `alloc_netdev()` est apparu dans Linux 3.17 pour indiquer d'où provient le nom du *device*. La plupart des drivers l'initialisent à `NET_NAME_UNKNOWN`.

La méthode `setup` passée en argument de `alloc_netdev()` sera invoquée pour initialiser les champs de la structure. Il existe des méthodes pré-définies comme `ether_setup()`.

Après l'appel `register_netdevice()`, l'interface devient accessible avec `ifconfig`.

## Activation d'interface

La méthode `open()` de la structure `net_device` est invoquée lorsque l'interface est activée par la pile IP. Il faudra à ce moment renseigner l'adresse matérielle de la carte, et démarrer le traitement par la pile IP.

```
void netif_start_queue (struct net_device *net_dev);  
void netif_stop_queue  (struct net_device *net_dev);
```

La fonction `netif_start_queue()` indique à la pile IP que la carte est disponible pour envoyer des paquets sur le réseau.

La fonction `netif_stop_queue()` indique un arrêt définitif de la carte.

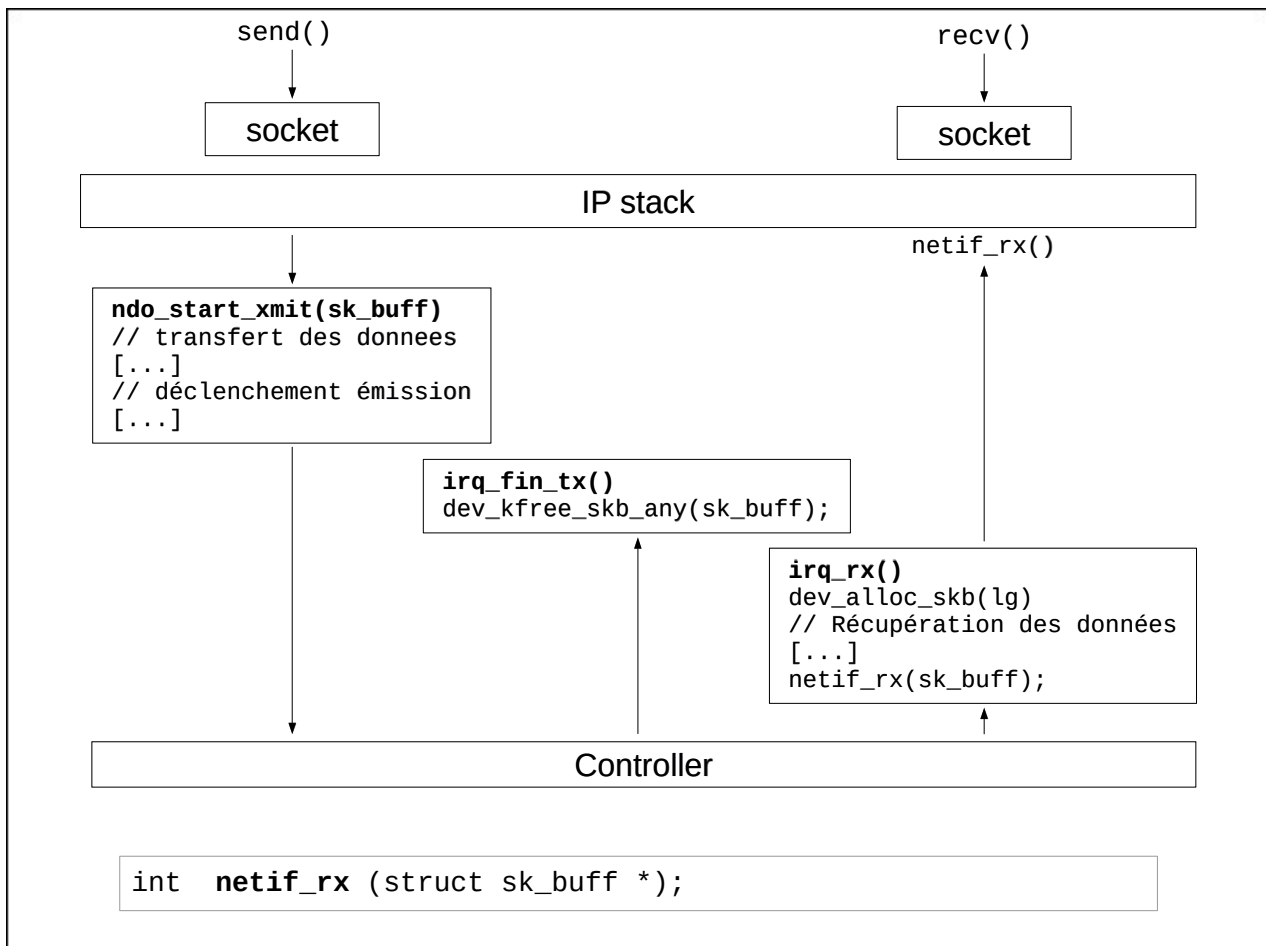
En cas de congestion sur le réseau, on peut désactiver temporairement les demandes d'émission de données par la pile IP.

```
void netif_tx_disable  (struct net_device *net_dev);  
void netif_wake_queue  (struct net_device *net_dev);
```

## Travaux pratiques : enregistrement d'un driver netdevice

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice VI-1 : enregistrement d'un driver *netdevice*.



Lorsque la pile IP désire émettre un paquet, elle appelle la méthode `ndo_start_xmit()` de la structure `net_device` en lui transmettant une structure `sk_buff` contenant le paquet. Cette méthode doit transférer les données au matériel et lui demander des les émettre.

Lorsque celui aura fini d'envoyer le paquet sur le réseau, il déclenchera une interruption. Le gestionnaire d'interruption appellera la fonction `dev_kfree_skb_any()` pour libérer la structure `sk_buf`.

Lorsqu'un paquet de données arrive depuis le réseau, le matériel doit déclencher une interruption. Le gestionnaire lira les données, et construira, avec la fonction `dev_alloc_skb()` une structure `sk_buff` autour des données. La structure `sk_buff` sera transmise à la pile IP en appelant `netif_rx()` qui représente le point d'entrée par le bas dans cette pile.

## Socket buffers

```
<skbuff.h>

struct sk_buff {
    struct net_device *dev;
    union {
        struct tcphdr  *th;
        struct udphdr  *uh;
        [...] } h;
    union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        [...] } nh;
    unsigned int        len;
    unsigned char        ip_summed;
    __be16              protocol;
    [...]
    unsigned char        *head, *data, *tail, *end;
};
```

```
struct sk_buff * dev_alloc_skb(unsigned int len);
```

L'argument *len* correspond à la longueur du bloc de données à réserver.

```
void dev_kfree_skb      (struct sk_buff *sk_b);
void dev_kfree_skb_irq (struct sk_buff *sk_b);
void dev_kfree_skb_any (struct sk_buff *sk_b);
```

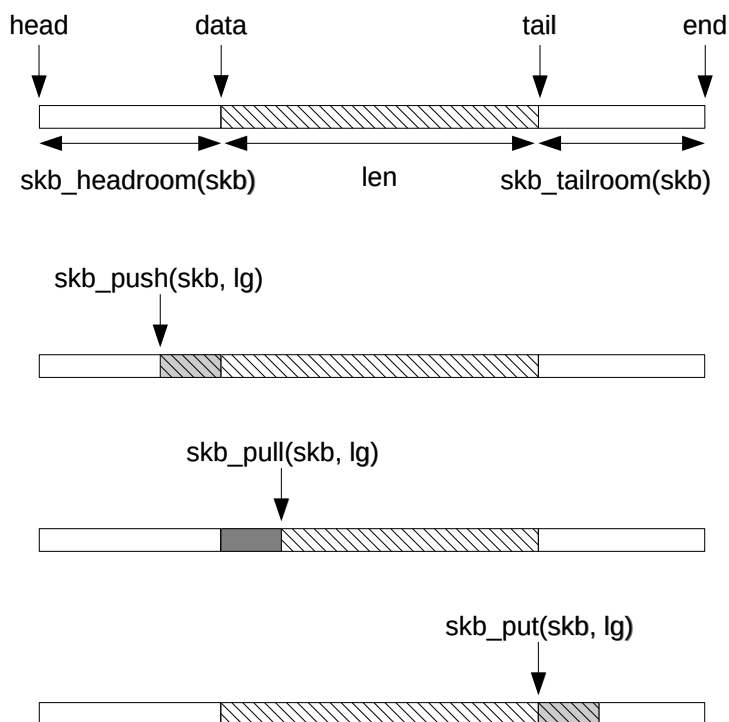
La structure `sk_buff` (*socket buffer*) est l'élément central de communication entre les couches de protocoles et les pilotes de périphériques. Cette structure incorpore des zones de données correspondant aux paquets à transmettre sur l'interface physique et des en-têtes pour les protocoles réseau.

L'allocation interne invoquée par `dev_alloc_skb()` utilise l'argument `GFP_ATOMIC` et peut donc être appelée dans un gestionnaire d'interruption. `dev_kfree_skb()` ne peut être appelé qu'en contexte d'appel-système, `dev_kfree_skb_irq()` en contexte d'interruption, et `dev_kfree_skb_any()` n'importe où.

Le champ `ip_summed` indique à la pile IP si une somme de contrôle a été vérifiée sur le paquet :

- `CHECKSUM_HW` : la *checksum* a été validée par l'interface matérielle ;
- `CHECKSUM_NONE` : la vérification doit être faite par la pile IP ;
- `CHECKSUM_UNNECESSARY` : la validation est inutile (périphérique virtuel par exemple).

Au retour de `dev_alloc_skb()`, les pointeurs `data` et `tail` sont identiques. La longueur `len` est disponible entre `tail` et `end`.



Pour remplir les champs `sk_b->pkt_type` et `sk_b->protocol`, on utilise :

```
<etherdevice.h>

__be16 eth_type_trans(struct sk_buff *,
                      struct net_device *);
```

Le type `__be16` indique un entier sur 16 bits qui ne sert qu'à des opérations binaires.

Pour dimensionner les zones de données, et ajouter ou extraire des en-têtes, on utilise principalement les routines suivantes :

```
unsigned char * skb_put (struct sk_buff *sk_b,  
                          int length);
```

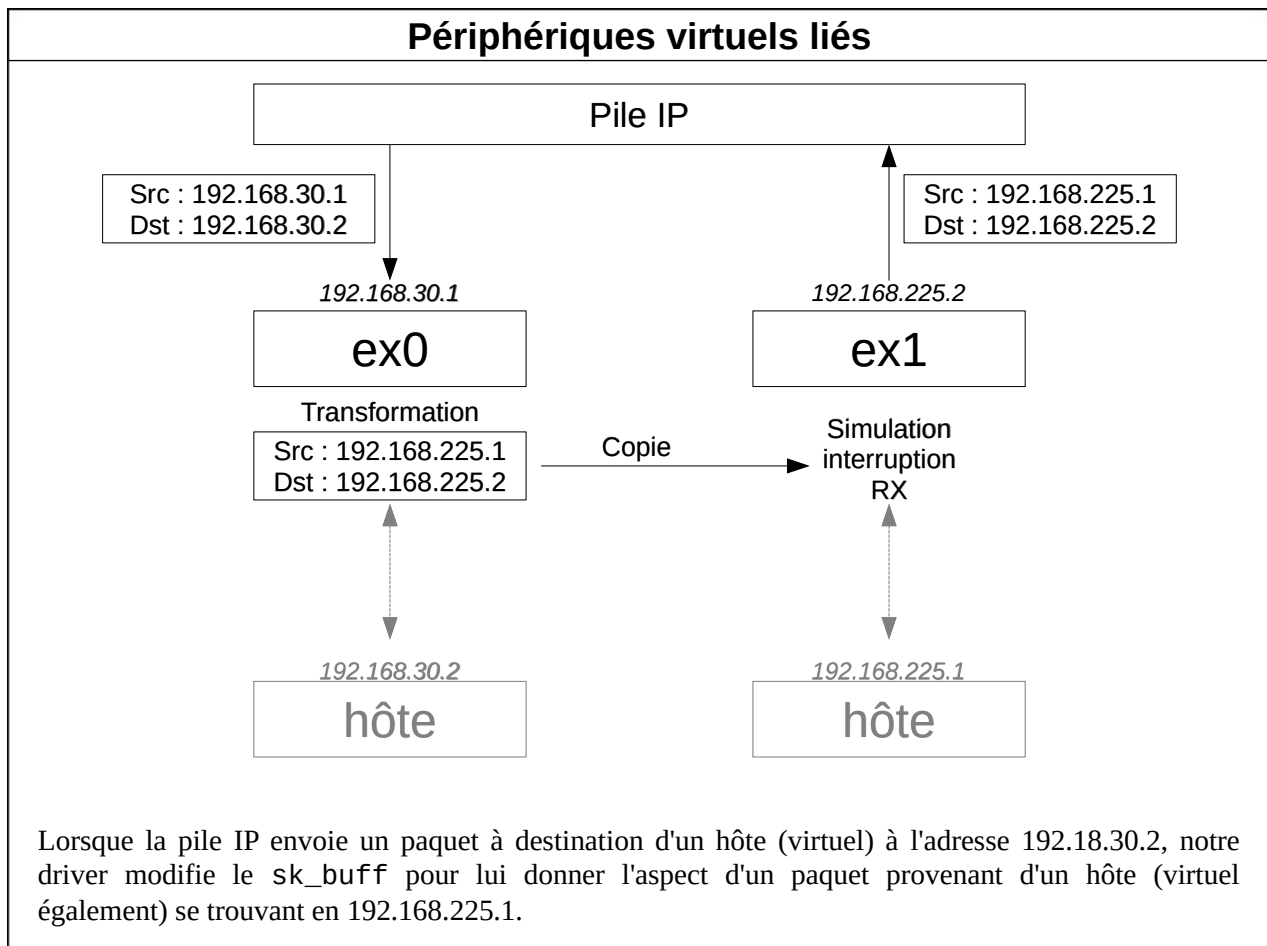
Augmente le pointeur *tail* et la taille *len* des données. Attention à ne pas dépasser la zone libre maximale – accessible avec *skb\_tailroom(sk\_b)*. Cette fonction renvoie un pointeur sur la zone nouvellement accessible.

```
unsigned char * skb_push (struct sk_buff *skb,  
                           int length);
```

Diminue le pointeur *data* et augmente la taille *len* des données. Attention à ne pas dépasser la zone libre maximale dans l'en-tête – accessible avec *skb\_headroom(sk\_b)*. Cette fonction renvoie un pointeur sur la zone nouvellement accessible.

```
void skb_pull (struct sk_buff *sk_b, int length);
```

Augmente le pointeur *data* et diminue la taille *len* des données.



Le paquet est ensuite copié en entrée de la seconde carte gérée par notre driver, et on simule le déclenchement d'une interruption de réception en appelant directement le gestionnaire.

Attention : ce principe (librement inspiré du livre [RUBINI 2005]) ne fonctionne qu'avec les protocoles de la pile IPv4 (modification directe de certains champs d'en-tête).

## Travaux pratiques : driver d'interfaces « miroirs »

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice VI-2 : driver d'interfaces « miroirs »

## Statistiques d'utilisation

```
struct net_device_stats
{
    unsigned long rx_packets; // total packets received
    unsigned long tx_packets; // total packets transmitted
    unsigned long rx_bytes;  // total bytes received
    unsigned long tx_bytes;  // total bytes transmitted
    unsigned long rx_errors; // bad packets received
    unsigned long tx_errors; // packet transmit problems
    unsigned long rx_dropped; // no space in linux buffers
    unsigned long tx_dropped; // no space available in kernel
    unsigned long multicast; // multicast packets received
    unsigned long collisions;
    [...]
}
```

```
struct net_device_ops {
    [...]
    struct net_device_stats * (*ndo_get_stats)(
                                                struct net_device*);
    [...]
}
```

La structure `net_device_stats` regroupe les informations statistiques concernant une interface. On peut les consulter – entre autres – avec `ifconfig`.

Les informations sont maintenues dans le driver et sont fournies lors de l'appel de la méthode `ndo_get_stats`. La structure de statistiques est généralement insérée dans le champ privé de la structure `net_device`.

## Travaux pratiques : statistiques d'utilisation d'une interface réseau

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice VI-3 : statistiques d'utilisation d'une interface réseau.