Les utilitaires relatifs aux namespaces

Rachid Koucha
[Ingénieur développement logiciel]

Après un premier article [1] sur la notion de namespace et de conteneur avec un passage en revue des appels système, ce second opus se concentre sur les commandes mises à disposition de l'utilisateur.

Table des matières

Avant-propos

Le code source des exemples utilisés dans cet article sont disponibles sur Github : https://github.com/Rachid-Koucha/linux_ns.

Cet article a été publié dans GNU Linux Magazine France n°240 du mois de septembre 2020 :

Introduction

En plus des appels système destinés aux programmeurs, des utilitaires sont à la disposition de l'opérateur pour la mise en oeuvre des namespaces. Ils recèlent des subtilités que nous allons découvrir et expliquer.

1 La commande unshare

Issue du paquet **util-linux**, cet utilitaire exécute un programme dans de nouveaux namespaces (cf. **man 1 unshare**) :

```
unshare [options] [program [arguments]]
```

C'est un enrobage de l'appel système **unshare()**. Les options sur la ligne de commande permettent de choisir les namespaces à créer (p. ex. **-p** pour un nouveau pid_ns, **-m** pour un nouveau mount_ns…).

Utilisons cette commande avec les options **-u**, **-p** et **-i** pour lancer un shell associé respectivement à de nouveaux namespaces UTS, PID et IPC (les droits du super utilisateur sont requis !) :

```
# unshare -u -p -i /bin/sh
$ date
mercredi 29 janvier 2020, 20:54:26 (UTC+0100)
$
```

Tout semble s'exécuter à merveille. On est cependant confronté à un problème étrange déjà évoqué, mais non encore expliqué dans l'article précédent. Nous arrivons à lancer un première commande comme **date** ci-dessus mais ensuite toute nouvelle commande se solde par une erreur de **fork()** :

```
$ ls
/bin/sh: 3: Cannot fork
$ date
/bin/sh: 7: Cannot fork
```

Si on lance une built-in du shell comme **echo** qui ne provoque pas de **fork()**/**exec()**, cela fonctionne par contre :

```
# echo $$
8840
```

Dans un autre terminal, listons les namespaces de ce nouveau shell :

```
# sudo lsns -p 8840
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    312     1 root /sbin/init splash
4026531836 pid       312     1 root /sbin/init splash
4026531837 user      307     1 root /sbin/init splash
4026531840 mnt       300     1 root /sbin/init splash
4026531992 net       307     1 root /sbin/init splash
4026532826 uts         1  8840 root /bin/sh
4026532937 ipc         1  8840 root /bin/sh
```

Nous constatons que le shell est bien associé à un nouvel ipc_ns et un nouveau uts_ns (créés par le processus **8840**) comme demandé mais son pid_ns est toujours le namespace initial (créé par le processus numéro **1**) ! En fait, lorsqu'un processus appelle **unshare()** pour créer et s'associer à un nouveau pid_ns ou **setns()** pour s'associer à un autre pid_ns, **ce sont ses fils qui seront effectivement associés au nouveau pid_ns**. Dans notre exemple, la commande **unshare** appelle le service **unshare(CLONE_NEWUTS|CLONE_NEWPID|CLONE_NEWIPC)** puis le service **execve()** pour exécuter **/bin/sh** (cf. figure 1).

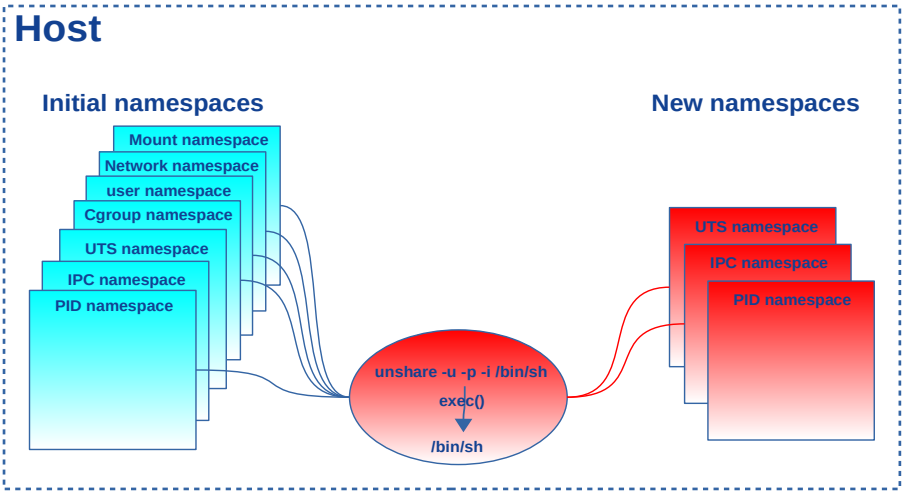

*Fig. 1 : **`unshare`** sans **`fork()`***

Comme on ne change pas de processus (c.-à.d. le programme **`unshare`** est « écrasé » par le programme **`/bin/sh`**), le shell ainsi lancé ne change pas de pid_ns. Seuls les nouveaux ipc_ns et uts_ns lui sont associés. Ce sont ses fils qui entreront dans le nouveau pid_ns. Lorsque la première commande **`date`** est lancée, un premier **`fork()`**/**`exec()`** est effectué pour l'éxécuter. Le processus fils résultant hérite bien de tous les namespaces de son père avec en plus l'association au nouveau pid_ns (cf. figure 2).

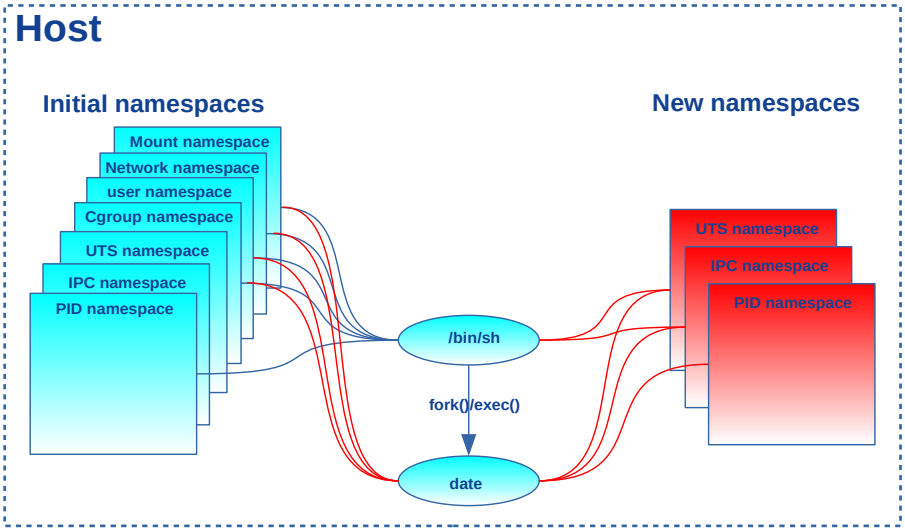

Fig. 2 : Exécution dans le nouveau pid_ns

Cette commande s'exécute bien dans le nouveau pid_ns avec l'identifiant **`1`**. Mais quand la commande se termine, le pid_ns disparaît avec le processus associé (cf. figure 3).

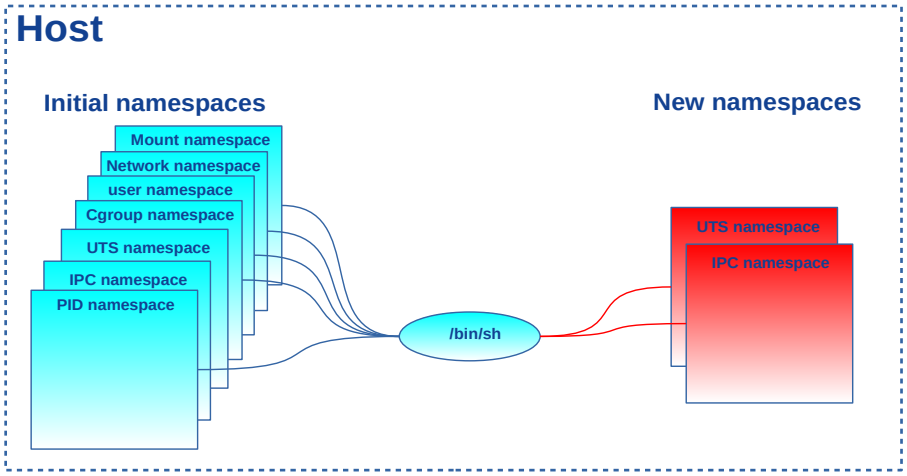

Fig. 3 : Disparition du nouveau pid_ns

Cela rend impossible par la suite la création de tout nouveau processus. D'où l'erreur sur le **fork()**. Pour s'en convaincre, terminons le shell avec la built-in **exit** et utilisons un petit script shell **alarm** qui sommeille le nombre de secondes passés en paramètre et affiche le message « !!!!! ALARM !!!!! » lorsqu'il se termine à échéance de la temporisation. Lançons-le en background comme premier processus du shell. Comme il ne se termine qu'à échéance du délai passé en paramètre, des commandes peuvent s'exécuter pendant le laps de temps (p. ex. **date**, **gcc**...). Par contre, dès que le script **alarm** finit son exécution, en tant que premier processus du pid_ns, il entraîne avec lui la fermeture du pid_ns. C'est alors que l'erreur de **fork()** est de retour au lancement de nouvelles commandes :

```
# unshare -u -p -i /bin/sh
# ./alarm 30 &
# date
mercredi 29 janvier 2020, 22:52:51 (UTC+0100)
# gcc --version
gcc (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# ps
  PID TTY          TIME CMD
 2431 pts/0    00:00:00 bash
12143 pts/0    00:00:00 sudo
12144 pts/0    00:00:00 sh
12575 pts/0    00:00:00 alarm
12576 pts/0    00:00:00 sleep
12597 pts/0    00:00:00 ps
# !!!!! ALARM !!!!!
[1] + Done                       ./alarm 30
# gcc --version
/bin/sh: 6: Cannot fork
```

La commande **unshare** recèle cependant des fonctionnalités très utiles afin d'éviter cette chausse-trape. L'option **-f** provoque un **fork()** et exécute le programme demandé dans un shell fils. C'est exactement ce dont nous avons besoin ici car le shell est exécuté dans un processus fils qui hérite des namespaces de son père et sera donc non seulement associé aux nouveaux uts_ns et ipc_ns mais aussi au nouveau pid_ns (avec l'identifiant **1**) comme indiqué en figure 4.

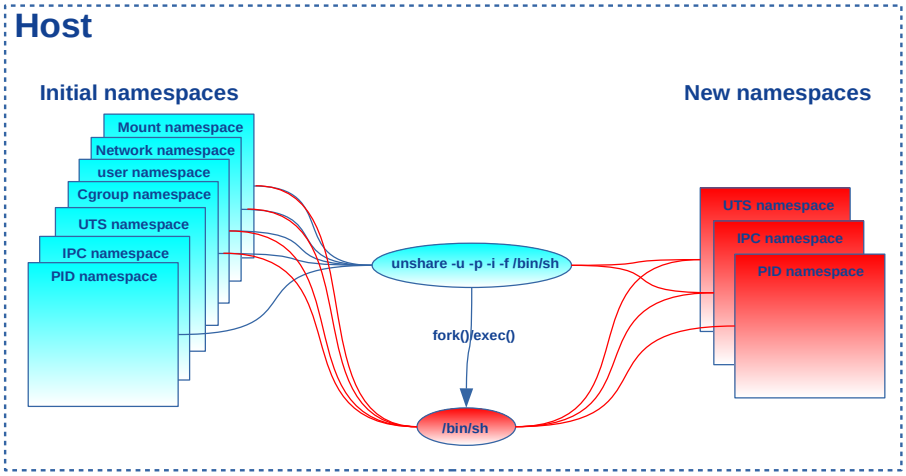

*Fig. 4 : **unshare** avec **fork()***

Le pid_ns reste actif tant que le shell n'est pas fini :

```
# exit
# unshare -u -p -i -f /bin/sh
# date
jeudi 30 janvier 2020, 07:33:49 (UTC+0100)
# gcc --version
gcc (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# echo $$
1
```

Nous vérifions bien que « echo $$ », résultat d'un **getpid()** dans le shell courant, affiche **1** comme identifiant car le shell s'exécute en tant que premier processus dans le nouveau pid_ns.

Restons dans ce shell et lançons la commande **ps** :

```
# ps
  PID TTY          TIME CMD
 2118 pts/0    00:00:00 bash
 3726 pts/0    00:00:00 sudo
 3728 pts/0    00:00:00 unshare
 3729 pts/0    00:00:00 sh
 3797 pts/0    00:00:00 ps
```

Les identifiants de processus affichés ne correspondent pas à ce qu'on pourrait s'attendre et notamment pour notre shell. En effet, il a le pid **3729** alors que « echo $$ » affiche **1**. En fait, **ps** affiche le pid du shell vu du côté pid_ns initial c'est-à-dire du pid_ns père dans la hiérarchie des namespaces. Alors que « $$ » est le pid vu du pid_ns du shell (le pid_ns fils).

Nous avons une fois de plus à faire à une subtilité des namespaces. Tandis que « echo $$ » obtient le pid dans le descripteur de tâche au sein du noyau, la commande **ps** va glaner ses informations dans le système de fichiers **/proc**. Ce dernier exporte des informations du noyau qui correspondent au pid_ns du processus qui a effectué son montage (cf. paragraphe « /proc and PID namespaces » dans **man 7 pid_namespaces**). Comme **/proc** a été monté au démarrage du système par un processus associé au pid_ns initial, il exporte donc des informations vues de ce namespace. Il faudrait remonter **/proc** dans ce shell pour obtenir les bonnes informations. Mais pour ne pas perturber le mount_ns initial, il est conseillé de le monter dans un nouveau mount_ns. C'est justement ce que proposent les options **-m** (création d'un nouveau mount_ns) et **--mount-proc** (remontage de **/proc**) de la commande **unshare** ! Sortons donc du shell et relançons-le avec ces options en supplément :

```
# exit
# unshare -u -p -i -f -m --mount-proc /bin/sh
# echo $$
1
# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 sh
    2 pts/0    00:00:00 ps
# date
vendredi 27 mars 2020, 20:29:59 (UTC+0100)
```

On ne voit donc plus que les deux seuls processus du pid_ns fils : le shell (pid **1** car il est le premier processu du namespace) et **ps** (pid **2** lancé en second).

2 La commande nsenter

Issue du paquet **util-linux**, cet utilitaire exécute un programme dans les namespaces d'autres processus (cf. **man 1 nsenter**) :

```
nsenter [options] [program [arguments]]
```

C'est un enrobage de l'appel système **setns()**. L'option **-t** sur la ligne de commande permet de spécifier un processus cible avec lequel on veut partager les namespaces spécifiés par les mêmes options que la commande **unshare** (e.g. **-m** pour mount_ns, **-u** pour uts_ns...).

Dans un terminal, affichons le nom du hôte (**rachid-pc**). Puis, relançons la commande d'illustration de la commande **unshare** vue au paragraphe précédent. L'invocation de **hostname** affiche le même nom de hôte que celui de l'uts_ns initial car il a été hérité. Dans ce nouvel uts_ns, on change le nom en **new-pc** :

```
# hostname
rachid-pc
# unshare -f -p -u -m --mount-proc /bin/sh
# echo $$
1
# hostname
rachid-pc
# hostname new-pc
# hostname
new-pc
```

Dans son pid_ns, le shell ainsi lancé a l'identifiant **1** car il est le premier processus dans son namespace. Mais il a un identifiant différent dans le pid_ns initial. Comme c'est le fils de la commande **unshare**, dans un autre terminal, la commande **ps** nous permet de l'obtenir :

```
# pidof unshare
29206
# ps -ef | grep 29206
root     29206 29203  0 22:36 pts/1    00:00:00 unshare -f -p -u -m --mount-proc /bin/sh
root     29207 29206  0 22:36 pts/1    00:00:00 /bin/sh
```

Donc le shell a l'identifiant **29207** dans le pid_ns initial. Dans ce même terminal, utilisons **nsenter** pour invoquer un autre shell dans les mêmes namespaces (pid_ns, uts_ns et mount_ns) :

```
# hostname
rachid-pc
# nsenter -t 29207 -p -u -m /bin/sh
# ps
  PID TTY          TIME CMD
    5 pts/0    00:00:00 sh
    6 pts/0    00:00:00 ps
# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 22:36 pts/1    00:00:00 /bin/sh
root         5     0  0 22:54 pts/0    00:00:00 /bin/sh
root         7     5  0 22:54 pts/0    00:00:00 ps -ef
# echo $$
5
# hostname
new-pc
```

Le nouveau shell a l'identifiant **5** dans le pid_ns cible. Cela confirme que la commande **ps** va chercher ses informations dans le système de fichiers **/proc** correspondant au pid_ns du processus qui l'a monté (le shell lancé par la commande **unshare** dans notre cas) et on vérifie bien qu'il est dans le même uts_ns en affichant le nom de hôte (**new-pc**).

3 La commande lsns

Issue du paquet **util-linux**, cet utilitaire parcourt le répertoire **/proc** et pour tous les identifiants de processus, relève et affiche les informations relatives aux namespaces (cf. **man 8 lsns**). Sans paramètres, il affiche par défaut le numéro d'inode (colonne **NS**), type de namespace (colonne **TYPE**), nombre de processus associés au namespaces (colonne **NPROCS**), plus petit identifiant de processus dans le namespace (colonne

PID), Nom d'utilisateur du processus (colonne **USER**), ligne de commande du processus (colonne **COMMAND**) :

```
# lsns
        NS TYPE   NPROCS   PID USER            COMMAND
4026531835 cgroup    328     1 root            /sbin/init splash
4026531836 pid       328     1 root            /sbin/init splash
4026531837 user      321     1 root            /sbin/init splash
4026531838 uts       314     1 root            /sbin/init splash
4026531839 ipc       318     1 root            /sbin/init splash
4026531840 mnt       306     1 root            /sbin/init splash
4026531860 mnt         1    57 root            kdevtmpfs
4026531992 net       317     1 root            /sbin/init splash
4026532340 mnt        12   400 root            /lib/systemd/systemd-udevd
4026532341 uts        12   400 root            /lib/systemd/systemd-udevd
[...]
```

Cependant, la manuel précise bien que l'affichage par défaut de la commande est sujet à des changements dans les futures versions. Il ne faut par conséquent pas s'y fier dans le cas d'une utilisation dans des scripts qui filtrent sa sortie. Il est conseillé d'utiliser **--output** suivie de la liste des colonnes à afficher. L'option **--help** permet d'obtenir tous les noms de colonnes acceptés :

```
# lsns --help
[...]
 -o, --output <list>    define which output columns to use
     --output-all       output all columns
[...]
Available output columns:
          NS  namespace identifier (inode number)
        TYPE  kind of namespace
        PATH  path to the namespace
      NPROCS  number of processes in the namespace
         PID  lowest PID in the namespace
        PPID  PPID of the PID
     COMMAND  command line of the PID
         UID  UID of the PID
        USER  username of the PID
     NETNSID  namespace ID as used by network subsystem
        NSFS  nsfs mountpoint (usually used network subsystem)
[...]
```

L'option **-p** est aussi très utile pour se focaliser sur les namespaces d'un processus donné. Reprenons notre exemple de shell lancé via la commande **unshare**. La commande **lsns** parcourant **/proc**. Vue du mount_ns du sous-shell, il n'y a que le shell (pid **1**) et la commande **lsns** elle même (d'où le **2** pour le nombre de processus dans la colonne **NPROCS**) :

```
# unshare -f -p -u -m --mount-proc /bin/sh
# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 sh
    2 pts/0    00:00:00 ps
# lsns
        NS TYPE   NPROCS PID USER COMMAND
4026531835 cgroup      2   1 root /bin/sh
4026531837 user        2   1 root /bin/sh
4026531839 ipc         2   1 root /bin/sh
4026531992 net         2   1 root /bin/sh
4026533107 mnt         2   1 root /bin/sh
4026533108 uts         2   1 root /bin/sh
4026533109 pid         2   1 root /bin/sh
```

Dans un autre terminal, **lsns** permet de récupérer le pid du sous-shell vu du pid_ns initial :

```
# lsns
[...]
4026533107 mnt         2  7450 root            unshare -f -p -u -m --mount-proc /bin/sh
4026533108 uts         2  7450 root            unshare -f -p -u -m --mount-proc /bin/sh
4026533109 pid         1  7451 root            /bin/sh
```

Nous vérifions ce que nous avons dit précédemment, à savoir que la commande **unshare** appelle certes le service système **unshare()** avec les options **CLONE_NEWNS**, **CLONE_NEWUTS** et **CLONE_NEWPID** pour respectivement créer un nouveau mount_ns, uts_ns et pid_ns mais seul le processus fils et sa descendance sont associés au nouveau pid_ns. D'où l'identifiant **7451** (commande **/bin/sh**) pour le premier processus du pid_ns et 7450 (commande **unshare**) pour le premier processus du mount_ns et de l'uts_ns. On peut ainsi afficher les informations sur les namespaces du sous-shell :

```
# lsns -p 7451
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    314     1 root /sbin/init splash
```

```
4026531837 user      306     1 root /sbin/init splash
4026531839 ipc       307     1 root /sbin/init splash
4026531992 net       306     1 root /sbin/init splash
4026533107 mnt         2  7450 root unshare -f -p -u -m --mount-proc /bin/sh
4026533108 uts         2  7450 root unshare -f -p -u -m --mount-proc /bin/sh
4026533109 pid         1  7451 root /bin/sh
# lsns -p 7451 -o NS,TYPE,PID
        NS TYPE     PID
4026531835 cgroup     1
4026531837 user       1
4026531839 ipc        1
4026531992 net        1
4026533107 mnt     7450
4026533108 uts     7450
4026533109 pid     7451
```

La colonne **NSFS** est le point de montage du « namespace file system » (absolument rien à voir avec **NFS**, le « network File System » !). Les namespaces sont régis par un système de fichiers virtuel interne au noyau. Nous verrons cela plus en détail dans un prochain article de cette série qui détaillera les mount_ns et dans la présentation de la commande **ip** qui suit.

La colonne **NETNSID** affiche l'identifiant de net_ns **NETLINK** **[2]** vu du net_ns du processus appelant. Si aucun identifiant n'a été assigné, la commande affiche « unassigned ». Nous reverrons cette notion dans la présentation de la commande **ip** qui suit.

4 La commande ip

L'utilitaire **ip** est le couteau suisse de la configuration réseau sous **GNU/Linux**. Il a rendu obsolète la fameuse commande **ifconfig**. Il est issue du paquet **iproute2**. Parmi ses nombreuses fonctionnalités, il ajoute un niveau d'abstraction sur les net_ns afin d'en faciliter la manipulation. Cela consiste à leur donner un nom (plus facile à mémoriser qu'un numéro d'inode ou un identifiant de processus !). Le manuel principal est dans **man 8 ip**. La partie concernant les options relatives aux namespaces a une description dédiée dans **man 8 ip-netns**.

4.1 Création d'un net_ns

La commande **ip** peut créér un nouveau net_ns tout en lui associant un nom. Ici nous créons le net_ns **newnet** avec la requête **add** :

```
# ip netns add newnet
```

La commande n'est pas très loquace. Pour connaître la liste des net_ns nommées par la commande **ip** :

```
# ip netns list
newnet
```

En interne, un net_ns est créé avec l'appel système **unshare(CLONE_NEWNS)**. Cependant, comme **un namespace disparaît lorsqu'il n'y a plus de processus qui lui est associé**, ce nouveau net_ns ne ferait pas long feu s'il disparaît lorsque la commande **ip** se termine. L'astuce est de créer un fichier dans **/var/run/netns/<nom_namespace>** puis d'appeler **unshare()** pour créer et s'associer à un nouveau mount_ns, puis enfin de monter le lien symbolique **/proc/<pid>/ns/net** qui caractérise le nouveau namespace sur le fichier créé dans **/var/run/netns** (on donnera plus de détails sur ce montage dans l'article concernant les mount_ns). Ce montage étant persistant à la fin de la commande **ip**, le net_ns reste actif bien qu'aucun processus n'y soit associé :

```
# ls -l /var/run/netns
total 0
-r--r--r-- 1 root root 0 mars   29 11:49 newnet
```

Quand le montage évoqué ci-dessus est réalisé, le fichier **mountinfo** dans **/proc** montre que le système de fichier virtuel **NSFS** interne au noyau est mis en oeuvre :

```
# cat /proc/$$/mountinfo
[...]
1160 26 0:23 /netns /run/netns rw,nosuid,noexec,relatime shared:5 - tmpfs tmpfs rw,size=1635172[...]
1183 1160 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
1184 26 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
```

Nous avions cité ce système de fichier lors de présentation de la commande **lsns**. Nous avions même vu que cette dernière affiche les point de montage **NSFS** lorsqu'ils sont mis en oeuvre. Mais à condition

d'exécuter **lsns** dans un contexte où un processus associé au net_ns considéré est visible. Dans notre cas de figure, il n'y a pas de processus associé à **newnet**. On peut utiliser l'option **exec** de **ip** pour lancer un processus en arrière-plan associé à ce net_ns :

```
# ip netns exec newnet sh -c 'sleep 300 &'
```

Ainsi la commande **lsns** sera à même de voir et afficher le point de montage **NSFS** correspondant au net_ns **newnet** :

```
# lsns -o ns,type,nprocs,pid,user,nsfs,command
        NS TYPE   NPROCS   PID USER            NSFS              COMMAND
[...]
4026532938 net         1  9111 root            /run/netns/newnet sleep 300
[...]
```

L'intérêt de créer un net_ns est d'isoler des interfaces réseau afin d'exécuter des commandes. Par défaut, un net_ns nouvellement créé n'a qu'une interface **loopback** :

```
# ip netns exec newnet ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

C'est d'un intérêt limité. Mais certaines interface réseau ont la possibilité de migrer d'un net_ns à l'autre. Comme chaque namespace a sa propre pile réseau, sa propre table de routage et ses propres règles de pare-feux, il est possible de mettre en place de multiples combinaisons de configurations réseau sur une même machine.

4.2 Attachement à un net_ns

La commande **ip** permet aussi de s'attacher à des net_ns existants. Par exemple, un net_ns mis en place pour un conteneur. Pour nous familiariser avec ce mécanisme, nous allons configurer une interface ethernet dans un conteneur **LXC**. Le conteneur nommé **bbox** est créé et démarré comme indiqué dans l'encadré « Mini-guide LXC » du premier article **[1]**. Par défaut, pour l'accès au réseau, le conteneur contient :

- Une interface loopback **lo** : Interface par défaut dans tout nouveau net_ns ;

- Une interface ethernet virtuelle **[3]** **eth0** : choisie par le template **busybox** et configuré par la commande **lxc-start** pour accéder au réseau internet.

L'interface ethernet virtuelle est déterminée par le paramètre **lxc.net** suivant dans le configuration du conteneur :

```
# cat /var/lib/lxc/bbox/config | grep lxc.net
lxc.net.0.type = veth
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
lxc.net.0.hwaddr = 00:16:3e:0e:f1:df
```

L'interface virtuelle ethernet est une sorte de tunnel. Dans le cadre de LXC, elle sert de lien entre le système hôte et le conteneur. Un client UDHCPC est aussi lancé pour configurer l'adresse IP sur cette interface (à l'autre extrémité, côté hôte, un serveur **dnsmasq** s'occupe des requêtes DHCP **[4]**).

```
# lxc-console -n bbox -t 0

BusyBox v1.30.1 (Ubuntu 1:1.30.1-4ubuntu4) built-in shell (ash)
Enter 'help' for a list of built-in commands.

bbox# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 00:16:3e:0e:f1:df brd ff:ff:ff:ff:ff:ff
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   14 root     /bin/udhcpc
[...]
```

Côté hôte, nous avons :

- **lo** : L'interface loopback associée au net_ns initial par défaut ;

- **eno1** : L'interface du port ethernet connectée à la box domestique ;

- **wlp6s0** : L'interface WIFI connectée à la box domestique ;

- **lxcbr0** : Le pont (bridge en anglais [5]) qui permet de connecter le lien ethernet virtuel hôte/conteneur au réseau. Les conteneurs connectés à ce bridge sont dans un sous-réseau. Un serveur **dnsmasq** [6] en écoute sur cette interface attribue les adresses IP (protocole DHCP) dans ce sous-réseau. Pour la communication avec l'extérieur via **eno1**, la distribution Linux se charge de la configuration de «l'IP forwarding » et des filtres réseau à partir du script **lxc-net** typiquement installé dans **/usr/libexec/lxc** ;

- **veth** : L'autre extrémité du tunnel ethernet virtuel reliant le hôte au conteneur.

```
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group
default qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
3: wlp6s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DORMANT group
default qlen 1000
    link/ether 00:08:ca:f5:89:9f brd ff:ff:ff:ff:ff:ff
4: lxcbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group
default qlen 1000
    link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
8: vethIQ0T55@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master lxcbr0 state UP
mode DEFAULT group default qlen 1000
    link/ether fe:d4:b7:23:e3:8b brd ff:ff:ff:ff:ff:ff link-netnsid 0
# ps -ef | grep dnsmasq
dnsmasq   5770  1680  0 13:52 ?        00:00:00 dnsmasq -u dnsmasq --strict-order --bind-interfaces
--pid-file=/run/lxc/dnsmasq.pid --listen-address 10.0.3.1 --dhcp-range 10.0.3.2,10.0.3.254 --dhcp-
lease-max=253 --dhcp-no-override --except-interface=lo --interface=lxcbr0
--dhcp-leasefile=/usr/local/var/lib/misc/dnsmasq.lxcbr0.leases --dhcp-authoritative
[...]
# brctl show
bridge name     bridge id               STP enabled     interfaces
lxcbr0          8000.00163e000000       no              vethIQ0T55
```

La configuration est schématisée en figure 5.

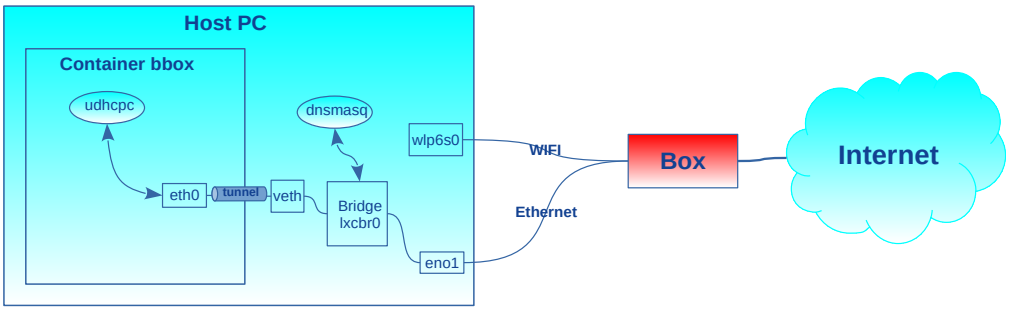

Fig. 5 : eno1 dans le net_ns du hôte

Avant de nous lancer, permettons-nous une nouvelle digression pour revenir sur la commande **lsns** et la colonne **NETNSID**. Nous avons dit qu'il s'agit d'un identifiant utilisé via le protocole **netlink**. En fait, un net_ns a la possibilité de mémoriser un identifiant par net_ns avec lesquels il est en intéraction. Lorsqu'on démarre un conteneur **LXC** avec une interface ethernet virtuelle, un identifiant de namespace (**NSID**) est attribué au net_ns du conteneur dans le net_ns courant. Ici il s'agit de l'identifiant **0** :

```
# ./lxc-pid bbox
5071
# lsns -p 5071 -o NS,TYPE,NETNSID,NSFS,NPROCS,PID,USER,COMMAND
        NS TYPE   NETNSID NSFS NPROCS   PID USER COMMAND
4026531837 user                  312     1 root /sbin/init splash
4026533164 mnt                     5  5071 root init
4026533165 uts                     5  5071 root init
4026533166 ipc                     5  5071 root init
4026533167 pid                     5  5071 root init
4026533170 net          0          5  5071 root init
4026533281 cgroup                  5  5071 root init
```

Pour en revenir à notre sujet, le but de la manipulation va consister à faire en sorte que le conteneur se connecte directement à l'extérieur sans passer par « le sous-réseau ethernet virtuel » mis en place par le template busybox de **LXC**. Cela implique de migrer l'interface ethernet **eno1** du net_ns du hôte vers celui du conteneur comme indiqué en figure 6. **Une interface réseau ne peut appartenir qu'à un seul net_ns**.

Comme nous le verrons plus tard certaines interfaces comme ethernet peuvent migrer d'un namespace à l'autre tandis que d'autres comme celles gérant le WIFI ne le peuvent pas.

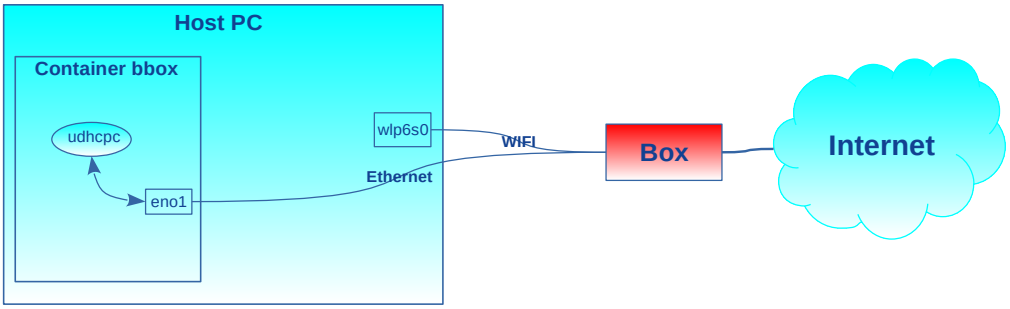

Fig. 6 : eno1 dans le net_ns du conteneur

Côté hôte, notons l'adresse IP sur l'interface **eno1** :

```
$ ip addr list eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.19/24 brd 192.168.0.255 scope global dynamic noprefixroute eno1
       valid_lft 78929sec preferred_lft 78929sec
    inet6 fe80::423f:cffd:687a:5713/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Désactivons l'interface ethernet virtuelle côté conteneur et vérifions que le trafic réseau ne passe plus à l'aide d'un **ping** vers le système hôte :

```
bbox# ip addr list eth0
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 00:16:3e:0e:f1:df brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.230/24 brd 10.0.3.255 scope global eth0
       valid_lft forever preferred_lft forever
bbox# ping 192.168.0.19 -c 2
PING 192.168.0.19 (192.168.0.19): 56 data bytes
64 bytes from 192.168.0.19: seq=0 ttl=64 time=0.118 ms
64 bytes from 192.168.0.19: seq=1 ttl=64 time=0.096 ms
[...]
bbox# ip link set eth0 down
bbox# ip addr list eth0
7: eth0@if8: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 00:16:3e:0e:f1:df brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.230/24 brd 10.0.3.255 scope global eth0
       valid_lft forever preferred_lft forever
bbox# ping 192.168.0.19 -c 2
PING 192.168.0.19 (192.168.0.19): 56 data bytes
ping: sendto: Network is unreachable
```

Côté hôte, nous associons un nom (e.g. **bbox_nsnet**) au net_ns du conteneur en utilisant l'identifiant du processus **init** de ce dernier. La commande **ip** permet en effet d'attacher un nom à un net_ns existant à partir de l'identifiant d'un processus associé au net_ns cible grâce à la requête **attach** :

```
# ./lxc-pid bbox
5071
# ip netns attach bbox_nsnet 5071
```

Dans ce cas, nous pouvons aussi voir le **NSID** du net_ns du conteneur via la requête **list** de **ip** :

```
# ip netns list
bbox_nsnet (id: 0)
newnet
```

Le net_ns **newnet** créé au paragraphe précédent n'a quant à lui pas de **NSID**.

L'attachement du net_ns du conteneur par la commande **ip** provoque aussi un montage du système de fichiers interne **nsfs** :

```
# ls -l /var/run/netns
total 0
-r--r--r-- 1 root root 0 mars  29 14:26 bbox_nsnet
-r--r--r-- 1 root root 0 mars  29 11:49 newnet
# lsns -p 5071 -o NS,TYPE,NETNSID,NSFS,NPROCS,PID,USER,COMMAND
        NS TYPE   NETNSID NSFS                NPROCS   PID USER COMMAND
4026531837 user                                  314     1 root /sbin/init splash
4026533164 mnt                                     5 10763 root init
4026533165 uts                                     5 10763 root init
4026533166 ipc                                     5 10763 root init
```

```
4026533167 pid                                      5 10763 root init
4026533170 net          0 /run/netns/bbox_nsnet     5 10763 root init
4026533281 cgroup                                   5 10763 root init
```

Nous désactivons l'interface ethernet du hôte puis nous la transférons dans le net_ns du conteneur (dans la couche d'abstraction de la commande **ip**, c'est **bbox_nsnet** configuré plus haut) :

```
# ip link set eno1 down
# ip addr list
[...]
2: eno1: <BROADCAST,MULTICAST> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
[...]
# ip link set eno1 netns bbox_nsnet
```

Nous constatons bien la migration de l'interface ethernet **eno1** dans le conteneur :

```
bbox# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
7: eth0@if8: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 00:16:3e:0e:f1:df brd ff:ff:ff:ff:ff:ff
```

Arrêtons le client **udhcpc** lancé au démarrage du conteneur et qui, par défaut (c.-à-d. sans paramètre), lance ses requêtes sur l'interface **eth0**. Puis relançons le pour qu'il adresse ses requêtes via l'interface ethernet **eno1** (option **-i**) tout juste transférée dans le conteneur et que nous prenons soin d'activer au préalable :

```
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   14 root     /bin/udhcpc
   15 root     /bin/getty -L tty1 115200 vt100
   16 root     /bin/sh
  114 root     {ps} /bin/sh
bbox# kill 14
bbox# ip link set eno1 up
bbox# ip link list
[...]
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
[...]
bbox# udhcpc -i eno1
udhcpc: started, v1.30.1
udhcpc: sending discover
udhcpc: sending select for 192.168.0.19
udhcpc: lease of 192.168.0.19 obtained, lease time 86400
```

L'affichage du client DHCP montre qu'il a obtenu l'adresse **192.168.0.19** de la part du serveur DHCP tournant dans la box domestique. La table de routage est mise à jour automatiquement et le client DHCP devient un daemon :

```
bbox# ip route show
default via 192.168.0.1 dev eno1
192.168.0.0/24 dev eno1 scope link  src 192.168.0.19
bbox# ip addr list
[...]
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.19/24 scope global eno1
       valid_lft forever preferred_lft forever
[...]
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   15 root     /bin/getty -L tty1 115200 vt100
   16 root     /bin/sh
  148 root     udhcpc -i eno1
  150 root     {ps} /bin/sh
[...]
```

Nous vérifions que le conteneur a bien accès à l'internet mondial avec un **ping** sur le serveur des **éditions Diamond** par exemple :

```
bbox# ping boutique.ed-diamond.com -c 2
PING boutique.ed-diamond.com (213.162.55.164): 56 data bytes
64 bytes from 213.162.55.164: seq=0 ttl=55 time=19.496 ms
```

```
64 bytes from 213.162.55.164: seq=1 ttl=55 time=25.170 ms

--- boutique.ed-diamond.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 19.496/22.333/25.170 ms
```

Tout se passe donc comme si deux machines indépendantes étaient connectées au réseau internet : le PC hôte via le WIFI **wlp6s0** et le conteneur **bbox** via l'interface ethernet **eno1** (comme schématisé en figure 6).

Arrêtons le conteneur :

```
bbox# <CTRL> + <a> + <q>
# lxc-stop -n bbox
# lxc-ls -f bbox
NAME STATE   AUTOSTART GROUPS IPV4 IPV6
bbox STOPPED 0         -      -    -
```

Si on liste les interface réseau disponibles dans le net_ns initial, on ne voit toujours pas l'interface **eno1** alors qu'une interface réseau doit être réaffectée au net_ns initial lorsque son net_ns disparaît :

```
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: wlp6s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 00:08:ca:f5:89:9f brd ff:ff:ff:ff:ff:ff
4: lxcbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
```

En fait, bien que stoppé, le net_ns du conteneur existe toujours car nous avons vu que **bbox_nsnet** est un point de montage du lien symbolique du net_ns du conteneur. Tant que ce montage est actif, le net_ns du conteneur continue à exister. Effectuons le « démontage » de **bbox_nsnet** à l'aide de la requête **del** de **ip** :

```
# cat /proc/$$/mountinfo
1160 26 0:23 /netns /run/netns rw,nosuid,noexec,relatime shared:5 - tmpfs tmpfs rw,size=1635172[...]
1183 1160 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
1184 26 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
426 1160 0:4 net:[4026533170] /run/netns/bbox_nsnet rw shared:235 - nsfs nsfs rw
427 26 0:4 net:[4026533170] /run/netns/bbox_nsnet rw shared:235 - nsfs nsfs rw
# ip netns del bbox_nsnet
# ip netns list
newnet
# cat /proc/$$/mountinfo
[...]
717 26 0:23 /netns /run/netns rw,nosuid,noexec,relatime shared:5 - tmpfs tmpfs rw,size=1635172[...]
1183 1160 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
1184 26 0:4 net:[4026532938] /run/netns/newnet rw shared:641 - nsfs nsfs rw
# ls -l /var/run/netns
total 0
-r--r--r-- 1 root root 0 mars  29 11:49 newnet
```

Maintenant, le net_ns du conteneur a disparu et l'interface ethernet **eno1** se retrouve de nouveau associée au net_ns initial :

```
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group
default qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
3: wlp6s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 00:08:ca:f5:89:9f brd ff:ff:ff:ff:ff:ff
4: lxcbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group
default qlen 1000
    link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
```

Avant de conclure cet article, on notera que pour avoir un conteneur **LXC** avec juste une interface **loopback** dans son net_ns, il suffit de modifier le paramètre de configuration **lxc.net** comme suit dans **/var/lib/lxc/bbox/config** :

```
[...]
lxc.net.0.type = empty
#lxc.net.0.type = veth
#lxc.net.0.link = lxcbr0
#lxc.net.0.flags = up
#lxc.net.0.hwaddr = 00:16:3e:0e:f1:df
[...]
```

Stoppons et relançons le conteneur :

```
# lxc-stop -n bbox
# ./lxc-start2 bbox
# lxc-console -n bbox -t 0

Connected to tty 0
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

BusyBox v1.30.1 (Ubuntu 1:1.30.1-4ubuntu4) built-in shell (ash)
Enter 'help' for a list of built-in commands.

bbox# ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Enfin, il est aussi possible de démarrer un conteneur LXC avec une interface ethernet en modifiant la configuration **/var/lib/lxc/bbox/config** comme suit :

```
[...]
lxc.net.0.type = phys
lxc.net.0.link = eno1
#lxc.net.0.type = veth
#lxc.net.0.link = lxcbr0
#lxc.net.0.flags = up
#lxc.net.0.hwaddr = 00:16:3e:0e:f1:df
[...]
```

Sortons de la console du conteneur, stoppons-le et redémarrons-le pour qu'il prenne en compte cette nouvelle configuration :

```
bbox# <ctrl> + <a> + <q>
# lxc-stop -n bbox
# ./lxc-start2 bbox
# lxc-console -n bbox -t 0

Connected to tty 0
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

BusyBox v1.30.1 (Ubuntu 1:1.30.1-4ubuntu4) built-in shell (ash)
Enter 'help' for a list of built-in commands.

bbox# ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
```

Pour obtenir une adresse IP, il faut réactiver l'interface (on aurait aussi pu décommenter **lxc.net.0.flags=up**) et redémarrer le client DHCP comme on l'a fait précédemment :

```
bbox# ip addr list
1: lo: <LOOPBACK> mtu 65536 qdisc noop qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
    7 root     /bin/getty -L tty1 115200 vt100
    8 root     /bin/sh
   11 root     {ps} /bin/sh
bbox# ip link set eno1 up
bbox# ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether c8:60:00:e3:b9:5e brd ff:ff:ff:ff:ff:ff
bbox# udhcpc -i eno1
udhcpc: started, v1.30.1
udhcpc: sending discover
udhcpc: sending select for 192.168.0.19
udhcpc: lease of 192.168.0.19 obtained, lease time 86400
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
    7 root     /bin/getty -L tty1 115200 vt100
```

```
    8 root     /bin/sh
   27 root     udhcpc -i eno1
   29 root     {ps} /bin/sh
bbox# ping boutique.ed-diamond.com -c 2
PING boutique.ed-diamond.com (213.162.55.164): 56 data bytes
64 bytes from 213.162.55.164: seq=0 ttl=55 time=33.822 ms
64 bytes from 213.162.55.164: seq=1 ttl=55 time=32.274 ms

--- boutique.ed-diamond.com ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 32.274/33.048/33.822 ms
```

Nous avons donc refait avec la configuration de **LXC** ce que nous avons fait manuellement en début de paragraphe.

Conclusion

Ce second article, à travers l'étude des utilitaires qui les mettent en oeuvre, a révélé et expliqué des subtilités inhérentes aux namespaces et les options qui permettent de les exploiter. Cela nous a aussi permis d'avoir un premier aperçu de la gestion des interfaces réseau dans les conteneurs **LXC**. Nous ne sommes pas au bout de nos surprises ! Dans le prochain article nous présenterons la vue des namespaces côté noyau.

Références

[1] R. Koucha, "Les namespaces ou l'art de se démultiplier", GNU/Linux magazine n°239, juillet/août 2020 : https://github.com/Rachid-Koucha/linux_ns/blob/main/articles/linux_namespaces_01.pdf

[2] Netlink : https://en.wikipedia.org/wiki/Netlink

[3] Paire ethernet virtuelle : http://man7.org/linux/man-pages/man4/veth.4.html

[4] Le protocole DHCP **:** https://fr.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol

[5] Les ponts réseau : https://en.wikipedia.org/wiki/Bridging_(networking)

[6] Le serveur dnsmasq : https://fr.wikipedia.org/wiki/Dnsmasq