

Isolation de processus par virtualisation de la couche réseau
dans un système open source compatible Windows.

Daniel Maxime

Mai 2014

Table des matières

1	Introduction	11
1.1	Présentation de l'entreprise	11
1.2	Cahier des charges	12
2	ReactOS	13
2.1	Historique	13
2.2	A propos de ReactOS et idéologie	13
2.2.1	Compatibilité	14
2.2.2	Sécurité	14
2.2.3	Légèreté	14
2.2.4	Libre	14
2.2.5	Confiance	15
2.2.6	Orienté objet	15
2.3	Technique	15
3	Le réseau dans ReactOS	17
3.1	NDIS (Network Driver Interface Specification)	17
3.2	Implémentation de la couche 3 (tcpip.sys)	19
4	Les namespaces	21
4.1	Virtualisation	21
4.1.1	Virtualisation noyau user-space	21
4.1.2	Hyperviseur de type 2	22
4.1.3	Hyperviseur de type 1	23
4.1.4	Virtualisation par Isolateur	23
4.2	ReactOS	24
4.3	Les network namespaces	25
4.4	Alternatives à l'isolation réseau sous d'autres systèmes	25
5	Implémentation des namespaces réseau dans ReactOS	27
5.0.1	Gestion des Processus	27
5.0.2	Implémentation du gestionnaire de namespaces réseau	28
5.0.3	Ajout et linkage d'un syscall dans le kernel	32
5.0.4	Unshare	33

5.1	Implémentation des namespaces dans la stack réseau (tcpip.sys)	34
5.1.1	Bibliothèque IP	35
5.1.2	Adresses	36
5.1.3	Interfaces	36
5.1.4	IP	37
5.1.5	Loopback	38
5.1.6	Voisinage	39
5.1.7	Routeur	41
5.2	Implémentation layer 2 et connexion au layer 3	43
5.2.1	Réception d'une trame dans le driver TCP/IP	45
5.2.2	Emission d'une trame depuis le driver TCP/IP	46
5.2.3	Virtual Ethernet	47
5.2.4	Implémentation d'un switch pour connecter les namespaces	50
5.3	Lien entre la couche utilisateur et le driver TCP/IP	56
5.3.1	Utilitaires de gestion : ipconfig.exe, route.exe, arp.exe, ping.exe,	56
5.3.2	IP Helper API (iphlpapi)	57
5.3.3	TDI (Transport Dispatch Interface)	58
6	Topologies de test	63
6.1	Overlapping d'adresses IP de destination	63
6.2	Overlapping d'adresses IP en communication inter-namespaces	64
7	Conclusion	65
7.1	Technique	65
7.2	Recommandations pour la suite	66
8	Conclusion personnelle	69
8.1	A propos de mes études	69
8.2	Sujet du TFE	69
8.3	Retour sur le stage	70
9	Remerciement	71
10	Informations complémentaires	73

11 Annexes	75
11.1 Environnement de développement et compilation	75
11.2 Exécution et débogage	76
11.3 lwIP (A Lightweight TCP/IP stack)	77
11.4 Syntaxe de la commande unshare.exe	79
11.5 Bloquage d'implémentation par le Plug'n'Play manager et NDIS Miniport	79
11.5.1 Interfaces type TAP	79
11.5.2 Interfaces virtuelles NDIS	80
11.6 Modifications collatérales	81
11.6.1 Mode Promiscuous dans le driver pcnet	81
11.6.2 Changer la couleur du Blue Screen	81
11.7 Comparaison des interfaces graphiques de gestion des protocoles	83

Table des figures

1	Capture d'écran de ReactOS 0.3.16 LiveCD dans VirtualBox	16
2	Couches NDIS	18
3	Couches TDI - NDIS	19
4	Diagramme d'une virtualisation en noyau user-mode	21
5	Diagramme de virtualisation avec un hyperviseur de type 2	22
6	Diagramme de virtualisation avec un hyperviseur de type 1	23
7	Diagramme d'une virtualisation par Isolateur	24
8	Schéma de deux namespaces réseau dans un même système	25
9	Diagramme de création d'un namespace en kernel-space	31
10	Diagramme de la couche d'appel du kernel à la Win32 API	32
11	Diagramme de relation avec une entrée dans la table ARP	40
12	Diagramme de recherche d'une route	42
13	Diagramme de lien entre LAN_ADAPTER et NDIS	45
14	Diagramme de réception d'un paquet depuis NDIS	45
15	Diagramme de préparation d'un paquet NDIS reçu	46
16	Diagramme de réception d'un paquet depuis NDIS vers TCP/IP	46
17	Diagramme d'émission d'un paquet depuis TCP/IP vers NDIS	47
18	Diagramme d'un lien classique entre IP_INTERFACE, LAN_ADAPTER et NDIS	48
19	Diagramme de lien avec une Virtual Ethernet (instance d'une LAN_ADAPTER + IP_INTERFACE)	49
20	Switching de trames	50
21	Diagramme montrant la nécessité du switch	52
22	Fonctionnement d'un ProtocolDriver NDIS	52
23	Diagramme d'utilisation de switch.sys	53
24	Pile d'appels depuis ipconfig jusqu'à tcpip.sys	61
25	Topologie de test avec overlapping d'adresses IP de destination	63
26	Topologie de test avec overlapping d'IP inter-namespaces	64
27	Enregistrement des drivers dans le kernel et dispatching d'actions	67
28	Couche réseau et utilisation de lwIP	78
29	Green Screen of Death	82
30	VirtualBox Bridged Networking Driver (NDIS ProtocolDriver)	83
31	Interface graphique non finalisée dans ReactOS	83

Table des extraits de code

1	Extrait de /drivers/network/tcpip/include/dispatch.h	30
2	Extrait de /include/psdk/winbase.h	31
3	Code ajouté à ntdll.spec, ntoskrnl.spec et kernel32.spec	33
4	Utilisation de « unshare » sous Linux	33
5	Nouveau namespace avec « unshare » sous Linux	34
6	struct IP_ADDRESS	36
7	struct IP_INTERFACE	37
8	struct LOOPLIST	39
9	struct NEIGHBOR_CACHE_ENTRY	40
10	struct FIB_ENTRY	41
11	struct LAN_ADAPTER	44
12	Prototype de la fonction NDIS s'occupant de la réception d'un paquet	51
13	struct SWITCH_NOTIFY	54
14	Reroutage des fonctions de réception et de transmission de paquets de tcpip.sys vers le switch virtuel	55
15	struct IP_ADAPTER_INFO (repris de la MSDN)	57
16	Adaptation de la liste d'entités TDI	59
17	Structure de données pour stocker plusieurs listes d'entités TDI	60
18	Fonctionnement de GetEntityList	60
19	Utilisation de unshare.exe	79
20	/drivers/network/dd/pcnet/	81

1 Introduction

1.1 Présentation de l'entreprise

La société Level IT a été fondée en décembre 2000 par deux ingénieurs civils (Hault Olivier et Laconte Luc) de l'Université de Liège. Son activité principale est axée sur le développement et la commercialisation de solutions informatiques permettant de répondre aux challenges et attentes de ses clients en termes d'excellence dans des domaines multiples comme la gestion de production, les Systèmes de Management Intégrés (SMI), la Total Productive Maintenance (TPM), ainsi que la gestion au quotidien des plans d'actions, d'exigences légales et de certifications (ISO, OSHAS, REACH, ...).

Avec ses produits phares (tel que Practeos, SAMS, Phone2Blog, ...) et grâce à ses principaux clients (ArcelorMittal, Tecteo, GlaxoSmithKline, Université de Liège, Nomacorc, ...), Level IT a su s'imposer comme référence en termes de développements de logiciels de qualité en Wallonie.

Son implantation stratégique au coeur du « Liège Science Park » vise à consolider ses activités de Recherches et Développement (R&D), tout en maintenant des relations étroites avec des acteurs de référence dans les domaines académiques, institutionnels, industriels et de services.

Leur principale technologie de développement repose sur Microsoft .NET mais ils ne se limitent pas à ça. En effet, en Recherche et Développement, une multitude de choix de langage de programmation différentes peuvent être utilisés pour répondre au mieux aux attentes des clients.



1.2 Cahier des charges

Certains systèmes d'exploitation, à l'heure actuelle, n'utilisent plus exclusivement la « virtualisation » à proprement parler, comme moyen d'isolement d'applications sur une même machine physique. Certains systèmes (comme Linux par exemple) permettent de lancer des applications dans différents namespaces, qui sont en fait une instanciation de différentes parties du système, permettant l'isolement de l'application. Pour ce qui est de la partie réseau, cela permet à différentes applications d'avoir différentes stack TCP/IP indépendantes l'une de l'autre. Pour l'instant, cela n'est pas possible nativement sur les systèmes Microsoft Windows.

Ce stage/TFE consiste à implémenter ce système de namespace réseau dans le kernel open-source de ReactOS qui fonctionne de la même façon que le kernel Microsoft Windows NT. Par exemple, on doit pouvoir lancer une application dans un namespace spécifique qui serait totalement indépendant des autres, pour avoir un isolement réseau pour l'application. Il faut que cet isolement puisse accéder au monde extérieur sans interférer avec les autres parties isolées. Il faut également pouvoir connecter plusieurs parties isolées ensemble.

Cet isolement permet d'avoir une table de routage, des interfaces réseaux, des règles de firewalling, etc. totalement indépendants vis-à-vis des autres namespaces sur le même système hôte.

Le cadre de ce projet sera une implémentation opensource. L'ajout de cette option doit ne pas interférer avec l'utilisation actuelle du système (il doit être optionnel et transparent pour l'application). Les applications déjà existantes doivent toujours pouvoir fonctionner de la même façon.

Actuellement, une solution propriétaire existe sous Microsoft Windows pour ça : Parallels Virtuozzo Containers (ou Parallels Containers for Windows). Cette solution requiert l'utilisation de la suite Parallels pour fonctionner. Pour les autres systèmes, des solutions tels que : OpenVZ, Linux-VServer, Solaris Containers, FreeBSD Jail, sysjail, HP-UX Containers,... sont des implémentations de virtualisation au niveau OS, mais présentent toutes différents avantages et inconvénients. Exemple : à l'heure actuelle sous Linux, à part OpenVZ, aucune solution ne propose une isolation totale des privilèges root.

Sous Windows, excepté « Parallels Containers », aucun système de virtualisation au niveau OS (iCore Virtual Accounts ou Sandboxie) ne supporte l'isolement de la stack réseau.

2 ReactOS

ReactOS est un système d'exploitation libre visant à être binairement compatible avec les systèmes d'exploitations propriétaires Microsoft Windows NT.

2.1 Historique

En 1996, des personnes ont formé un groupe appelé « FreeWin95 », visant à implémenter un clône de Windows 95. Cependant, suite à de trop longues discussions à propos de l'implémentation du système, le projet n'a absolument pas abouti.

Fin 1997, Jason Filby, un développeur d'Oracle (Durban, Afrique du Sud) alors chef coordinateur du projet FreeWin95, fait une demande générale sur la mailing list pour demander de relancer le projet. Il a été décidé de cibler Windows NT et d'obtenir des résultats plutôt que de parler indéfiniment sur comment faire le système. Leur but était réellement d'écrire du code pour avoir quelque chose de fonctionnel.

Le projet a été renommé en « ReactOS » suite à l'insatisfaction du monopole de Microsoft sur le marché des systèmes d'exploitation.

En février 1998, ReactOS est né. La première version disponible dans le svn est la 0.0.1, elle comporte 12000 lignes de code C (et 30000 lignes de headers). On y retrouve un noyau très léger mais avec déjà une arborescence similaire à l'actuelle. Cette version ne contient qu'un noyau et un boot loader.

Par la suite, les débuts de ReactOS ont été difficiles et lents. Seuls quelques personnes savaient comment écrire un noyau (en effet, ce n'est pas à la portée de tout le monde d'écrire un scheduler, un boot loader, etc.). Une fois le noyau un peu plus complet et stable, des drivers de base tels que les drivers IDE ou simplement un driver clavier ont pu être écrits et de plus en plus de personnes ont pu contribuer et commencer à coder à grande échelle pour le système.

2.2 A propos de ReactOS et idéologie

ReactOS est un système d'exploitation libre et open source basé sur l'architecture de Windows NT, supportant les logiciels et pilotes existant pour Windows.

ReactOS n'est pas basé sur Linux (comme l'est le projet WINE), cependant WINE est également implémenté dans ReactOS et constitue même la majorité du mode utilisateur du système. A l'heure actuelle, ReactOS est destiné aux développeurs plutôt qu'à des utilisateurs finaux, mais à terme, le système devrait être utilisable par n'importe qui, de la même façon qu'un Windows actuel. Leur but est de pouvoir faire un système binaires compatible avec Windows, sans être forcé de suivre la démarche commercial de Microsoft, ainsi que leurs idées (par exemple, ne pas forcer l'utilisation de Metro).

2.2.1 Compatibilité

« Change your OS, not your software ! » est un des slogans de ReactOS. L'idéologie du groupe est qu'on ne devrait pas changer de logiciels sous prétexte qu'on change de système d'exploitation. C'est pourquoi ils veulent faire fonctionner les logiciels et pilotes Windows sur leur système libre.

2.2.2 Sécurité

Contrairement à ce qu'on pourrait penser, le noyau NT a une très bonne sécurité de par sa conception. Il utilise des listes de contrôle d'accès évolués et est flexible. La mauvaise réputation qui a été apportée avec Windows XP principalement est due au fait que pour garder une rétro-compatibilité et pour aider les gens à migrer de Windows 9x vers Windows XP, il a fallu désactiver par défaut une grosse partie de la sécurité.

ReactOS a été prévu pour prendre directement l'aspect sécurisé de Windows NT.

2.2.3 Légèreté

ReactOS est conçu pour être léger et puissant. L'interface graphique se veut simple et légère comme celle de Windows 95 tout en gardant l'efficacité et les nouvelles fonctionnalités d'un Windows récent.

2.2.4 Libre

Le code source de ReactOS est disponible sous licence GNU GPL et sous BSD. La possibilité de voir le code source du noyau est facilement accessible via l'SVN et sa modification l'est aussi une fois inscrit sur le site principal de ReactOS.

2.2.5 Confiance

Depuis 1996, ReactOS a été écrit « from scratch », c'est à dire depuis rien. C'est une réimplémentation solide du noyau NT, prévu aussi bien pour l'embarqué que pour les ordinateurs personnels et serveurs. De plus, ReactOS prend en charge plusieurs méthodes d'implémentations basées sur d'autres familles d'OS tel que UNIX, VMS et OS/2 (exemple, il y'a une implémentation de la couche IP qui prend en charge les sockets BSD, ça veut dire qu'une application codée pour utiliser les sockets BSD peut potentiellement tourner sous ReactOS).

2.2.6 Orienté objet

ReactOS n'est pas un système orienté objet au sens propre du terme, mais il utilise des objets pour la représentation interne du système (comme Windows NT, ce concept sera expliqué plus loin lors de la communication entre le driver et le kernel). ReactOS n'est pas « pour » l'idéologie UNIX qui dit que « tout est fichier ». Bien que cela fasse la force de UNIX, c'est également le goulot d'étranglement d'après les développeurs et l'idéologie du groupe ReactOS (je n'ai pas trouvé de source à ce propos mais le code source ne montre en effet pas beaucoup d'allusion au fonctionnement de UNIX).

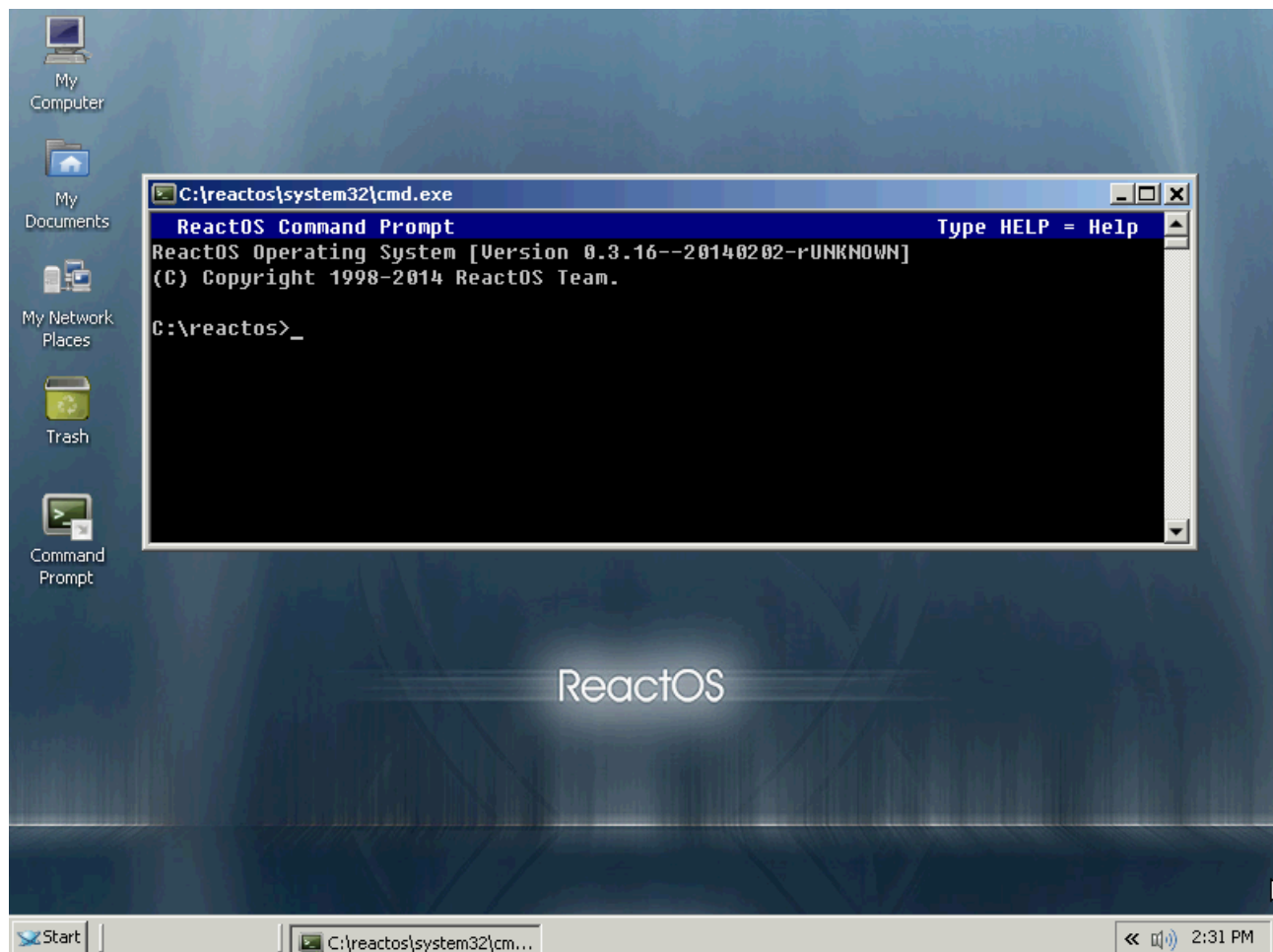
Pour ma part, je ne partage pas cet avis à propos du goulot sous UNIX.

2.3 Technique

ReactOS respecte assez bien, d'un point de vue logique, le fonctionnement d'un noyau NT, mais l'implémentation qui en est faite n'est pas toujours identique ou optimale. L'un des message important que les développeurs font passer lors d'un développement dans le système est : faire les choses bien est un bon début, mais les résultats sont plus importants que les performances. Le côté optimisation pourra être fait après par quelqu'un d'autre, le but premier est d'avoir quelque chose de fonctionnel.

Une fois qu'on commence à regarder le code source, on comprend que c'est clairement ça. Le code est loin d'être optimisé, mais il est fonctionnel. Une grosse partie n'est pas commentée et la documentation n'est pas forcément facile à trouver non plus. Un « doxygen » (générateur de documentation C++) est disponible pour le code, mais il n'est pas assez bien utilisé pour pouvoir s'en servir de référence pour le code.

FIGURE 1 – Capture d'écran de ReactOS 0.3.16 LiveCD dans VirtualBox



3 Le réseau dans ReactOS

La partie réseau de ReactOS n'est pas la partie du système la plus mise à jour. La couche TCP/IP¹ a initialement été écrite en 2000 (en se référant aux dates de modifications en commentaires dans le code) et n'a pas vraiment été mise à jour depuis. On se retrouve donc avec du vieux code mais fonctionnel. A noter aussi que seul le strict minimum a été implémenté comme cela sera montré plus loin.

ReactOS utilise le même fonctionnement global que Windows XP. Le principal repose sur NDIS. Ensuite, on retrouve une bibliothèque IP (couche 3) et une implémentation de lwIP (utilisée partiellement).

3.1 NDIS (Network Driver Interface Specification)

NDIS est une API pour les cartes réseaux écrite conjointement par Microsoft et 3Com et principalement utilisée dans les systèmes Microsoft Windows. Le projet est propriétaire et fermé, nous n'avons donc pas accès aux sources de l'implémentation, mais des versions opensource ont été faites par Reverse Engineering et il est donc possible d'utiliser des drivers NDIS sous Linux, FreeBSD, etc.

NDIS dispose de trois grandes parties :

- Miniport Drivers : ces drivers implémentent le fonctionnement hardware (ou logique en cas de cartes virtuelles) des cartes réseaux. C'est là que les interruptions matérielles, etc. sont gérées.
- ProtocolDrivers : ces drivers peuvent se binder à un ou des Miniports et communiquer avec eux de façon standardisée avec l'API.
- Intermediate Drivers : ces drivers se situent entre les Miniports et les Protocols Drivers. Ils permettent de filtrer le contenu qui y transite.

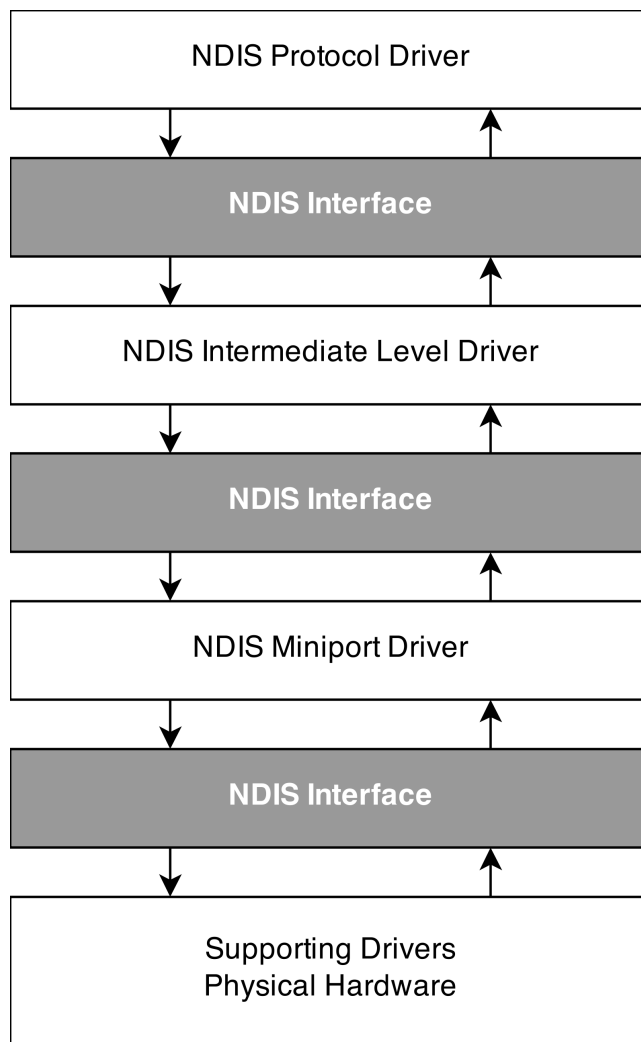
Typiquement, les drivers réseaux sont donc des Miniports, le driver TCP/IP est un Protocol-Driver et les Intermediates Drivers font office de Firewall, de Load Balancing Failover ou encore de translation entre des vieux transports et un driver Miniport qui ne le supporterait pas.

ReactOS utilise bien ce système de fonctionnement et implémente NDIS 5 (les versions 6 et

1. Par abus de langage, on emploie le terme « stack TCP/IP » pour l'entièreté de la couche réseau dans un système d'exploitation, mais en réalité dans le cadre de ce travail, seul la « stack IP » est mise en cause, en effet nous n'irons pas plus haut que la couche 3 du modèle OSI. Nous verrons cela plus tard.

supérieures ne sont pas supportées). Le système dispose de 3 drivers par défaut : ne2000, pcnet et rtl8193. Un grand nombre de drivers binaire faits pour Windows sont fonctionnels dans ReactOS, une liste des drivers testés est disponible sur le Wiki².

FIGURE 2 – Couches NDIS

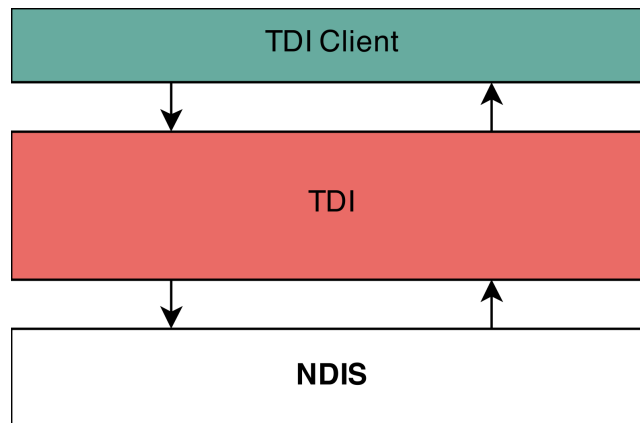


2. https://www.reactos.org/wiki/Supported_Hardware/Network_cards

3.2 Implémentation de la couche 3 (tcpip.sys)

Le driver `tcpip.sys` (dont les sources se trouvent dans `/drivers/network/tcpip/`) contient tout ce qui est nécessaire à la réception d'un paquet NDIS et la gestion interne des différentes tables (routing (RIB), forwarding (FIB), ARP, ...). Le driver TCP/IP à proprement parler est un ProtocolDriver NDIS. `tcpip.sys` est lié (statiquement³) à une bibliothèque logicielle qui s'appelle « ip » (source `/lib/network/ip`). Cette bibliothèque s'occupe de garder en mémoire toutes les informations de couche 3 pour le système (listes des adresses ip, les interfaces, leur binding avec NDIS, etc.). Des précisions l'implémentation de la `libip` sont à la section 5.1 page 35. Pour faire communiquer le ProtocolDriver avec le user-mode plus haut, le système utilise « TDI » pour faire le lien entre les deux. TDI est expliqué section 5.3.2 page 57.

FIGURE 3 – Couches TDI - NDIS



3. Cette précision est importante pour la suite

4 Les namespaces

Dans le sens large du terme, un « namespace » (espace de nom) est l'isolation d'un élément ou d'un groupe d'éléments, par un nom. On retrouve cette notion aussi bien dans la programmation qu'ailleurs, comme ici, dans une isolation système.

4.1 Virtualisation

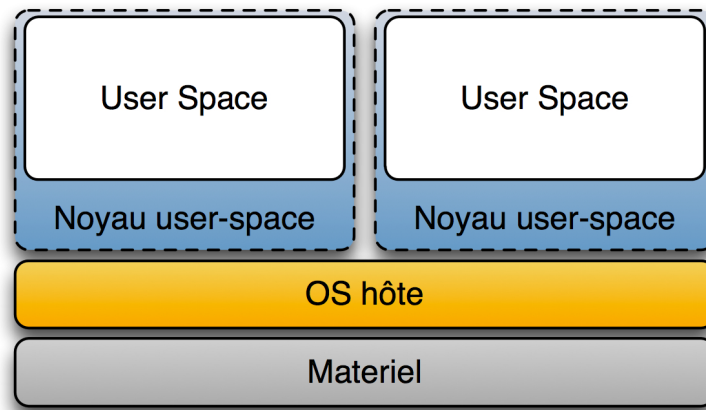
Si la virtualisation est très à la mode ces temps ci, c'est parce qu'elle apporte un réel gain de rentabilité du matériel dont on dispose. En effet, la virtualisation permet de partager les ressources d'une machine pour plusieurs utilisateurs, programmes, systèmes d'exploitation, ...

Cependant le terme « Virtualisation » est très vague. Il existe plusieurs techniques de virtualisation :

4.1.1 Virtualisation noyau user-space

Ce procédé permet de lancer un noyau par dessus un noyau existant, sans l'isoler. Chaque noyau a son propre espace utilisateur, comme une application classique, la seule isolation est celle qui se trouve à l'intérieur de cet espace utilisateur. Cette solution n'est vraiment utile que lors du développement d'un noyau pour pouvoir le tester plus facilement (jusqu'à une certaine limite, en effet vu comme ça, le noyau ne dispose pas d'un accès matériel comme un vrai noyau l'aurait).

FIGURE 4 – Diagramme d'une virtualisation en noyau user-mode



4.1.2 Hyperviseur de type 2

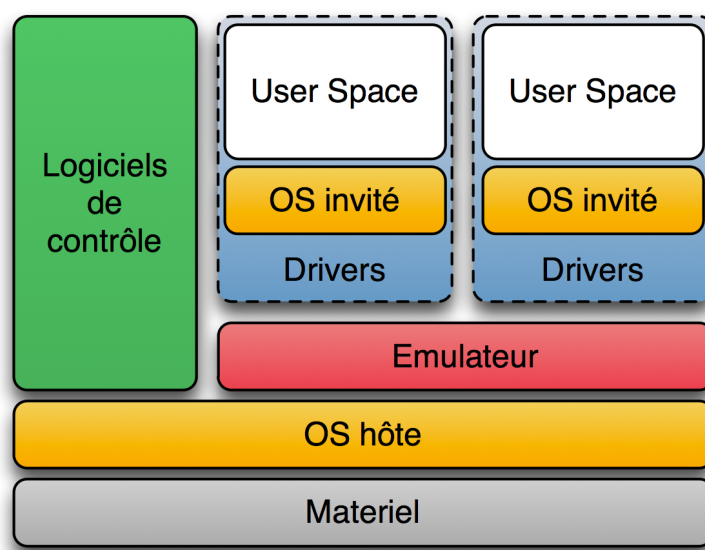
Un hyperviseur de type 2 est un logiciel qui tourne sur le système d'exploitation hôte et permet de lancer un ou plusieurs systèmes d'exploitation invités. Le logiciel virtualise (ou émule) le matériel pour les systèmes invités. Ces systèmes pensent alors dialoguer directement avec le matériel alors qu'en réalité il y a une couche entre les deux.

Les solutions les plus connues sont VMware Player/Workstation, Oracle VirtualBox ou encore Parallels Desktop.

La différence entre l'émulation et la virtualisation est l'accessibilité du système hôte. Un émulateur s'occupe de transférer les demandes du système invité vers le système hôte. L'avantage est côté performances vu qu'il ne s'agit que d'une translation de données, le problème est que le système invité doit être compatible avec le matériel du système hôte.

Pour faire tourner des systèmes totalement différents et même non compatibles avec le matériel hôte (exemple faire tourner un système embarqué ARM ou MIPS sur du x86), il faut passer par un émulateur. Son job consiste, lui, à traduire tout ce qui passe entre le système invité et hôte. L'impacte sur les performances est très important au vu du nombre d'opérations qu'il faut faire. Un émulateur libre très connu est [qemu](#). Cet émulateur (qui fait aussi de la virtualisation) a été utilisé dans ce projet car il propose des outils de débogage très avancés.

FIGURE 5 – Diagramme de virtualisation avec un hyperviseur de type 2

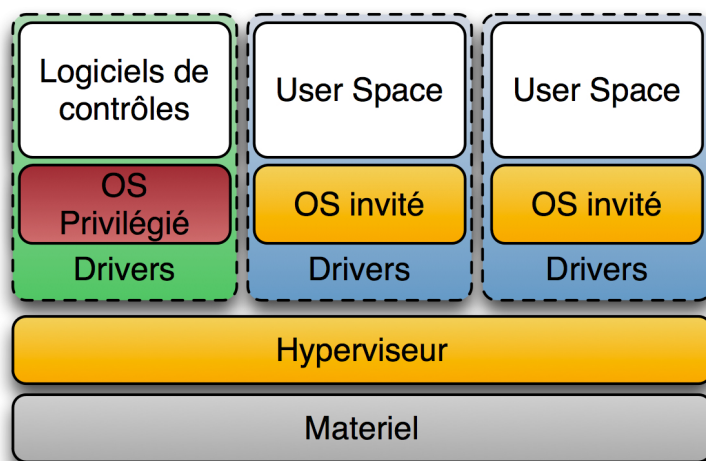


4.1.3 Hyperviseur de type 1

Un hyperviseur de type 1 est comme un noyau très léger et dédié pour faire tourner des machines virtuelles dessus. L'avantage d'un système dédié est qu'il est prévu et optimisé pour, la gestion des ressources est grandement mieux gérée par ce système et permet des performances très élevées. Ce type d'hyperviseur n'est du coup pas prévu pour tourner sur une machine end-user mais bien dédié sur un serveur qui ne fait que ça. C'est actuellement la solution la plus aboutie pour virtualiser entièrement un système d'exploitation complet en dégradant le moins les performances.

Cependant, l'utilisation d'un noyau très léger a pour contrainte de n'être compatible qu'avec un nombre restreint de matériel et ces solutions sont généralement onéreuses. Heureusement, des solutions libres existent comme Citrix Xen, KVM, ... Côté propriétaire, les plus connus sont Microsoft Hyper-V, VMware ESX.

FIGURE 6 – Diagramme de virtualisation avec un hyperviseur de type 1



4.1.4 Virtualisation par Isolateur

Un isolateur est un logiciel qui isole l'exécution d'un programme dans un contexte bien précis. L'isolateur permet de ne montrer que ce qu'il veut à l'application qui tourne derrière, c'est lui qui s'occupe de filtrer le monde réel à l'application qu'il englobe. De ce fait on peut faire tourner plusieurs instances d'une application sans que l'une n'interfère avec l'autre.

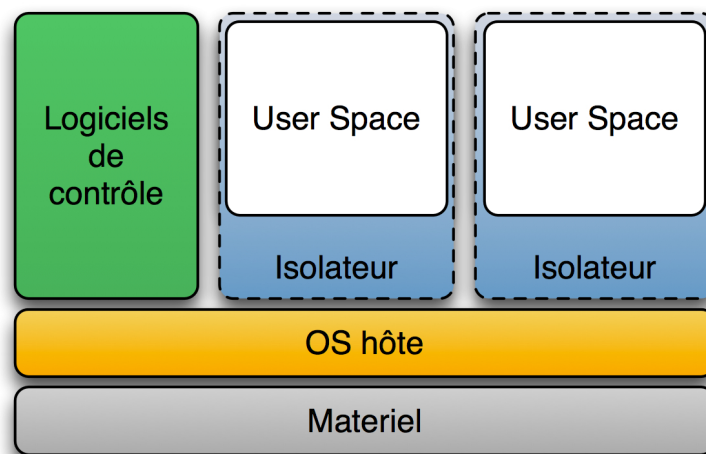
Niveau performance, cette technique est très rentable car l'overhead (le temps perdu à se gérer

soi-même) est très faible. Il n'y a qu'un seul noyau qui tourne, qu'une seule gestion de la machine globale et des petites parties qui s'occupent d'isoler.

Sous Linux, c'est effectué via les Namespaces Linux. Il dispose de plusieurs formes et permet d'isoler différentes parties du système indépendamment. On peut par exemple n'isoler que le réseau, que les processus, que les systèmes de fichiers, etc.

Sous BSD, ce procédé s'appelle BSD Jail, sous Solaris, ce sont des « zones ». A l'heure actuelle, sous Windows il n'y a qu'une solution propriétaire développée par Parallels qui s'applique comme surcouche sur le système d'exploitation.

FIGURE 7 – Diagramme d'une virtualisation par Isolateur



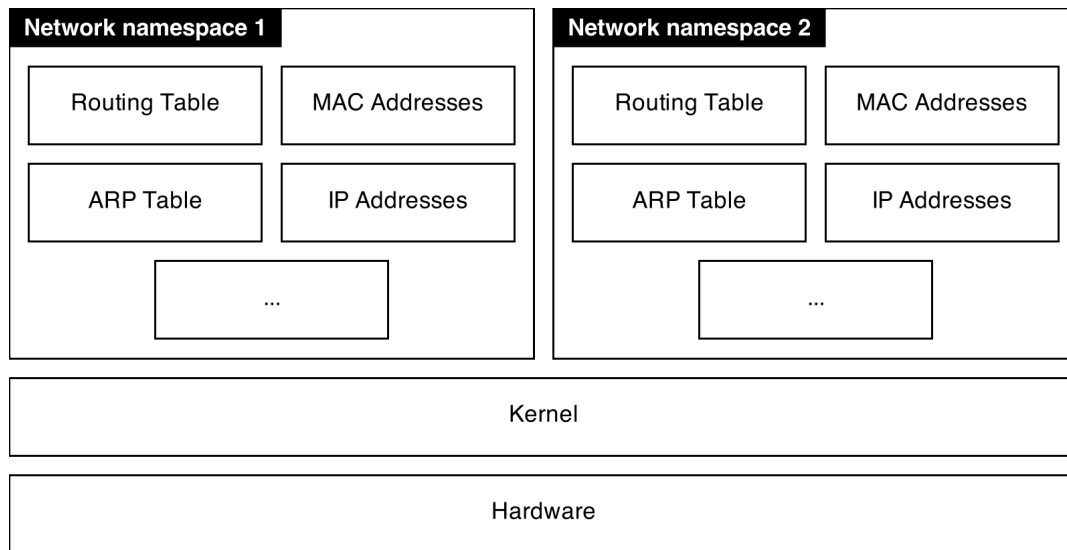
4.2 ReactOS

C'est cette dernière technique qui a été retenue pour mon implémentation dans ReactOS. De la même façon que sous Linux, nous allons isoler la partie réseau du système pour permettre de lancer plusieurs applications (identiques ou différentes) dans des contextes distincts.

4.3 Les network namespaces

Un « namespace réseau » est un contexte réseau complet isolé. Un contexte réseau complet prend réellement toutes les notions de réseaux d'une stack TCP/IP générale : table de routage, table ARP, interfaces IP, cartes réseaux, couche ethernet, etc.

FIGURE 8 – Schéma de deux namespaces réseau dans un même système



En d'autres termes, avoir plusieurs namespaces réseau dans un même système consiste à instancier plusieurs fois la stack TCP/IP sans aucun liens (au départ en tout cas, rien n'empêche la connexion des namespaces entre eux, comme on verra plus loin) entre ces différentes instances.

4.4 Alternatives à l'isolation réseau sous d'autres systèmes

Sous Windows, la seule solution qui permette cette approche est la solution propriétaire « Parallels Virtuozzo Containers », hélas il n'y a que très peu d'informations sur le fonctionnement de son implémentation.

Sous Cisco, ce principe est appelé « VRF » (Virtual Routing and Forwarding), il consiste à avoir plusieurs tables de routage différentes qui co-existent sur un même routeur.

Sous Solaris (depuis Solaris 10, 2005) on retrouve une isolation « Solaris Containers » qui inclut ce qu'ils appellent des « Zones ». Chaque zone a son propre nom (node name), sa propre stack réseau et son propre stockage.

Sous FreeBSD, la technique d'isolation se nomme « FreeBSD Jails », cependant cette isolation ne crée pas une nouvelle stack réseau. Il n'y a qu'une protection (isolation) aux niveaux processus, fichier et des sessions utilisateur/administrateur.

Pour expliquer comment j'ai implémenté l'isolation de la stack réseau dans ReactOS, je vais expliquer le fonctionnement interne de la stack et comment j'ai isolé les différentes parties qui en dépendent (processus, etc.). Je vais commencer par les processus, puis les différentes couches OSI intéressantes (layer 3 puis layer 2) pour enfin finir sur l'implémentation du lien entre les couches supérieures du système (le user-mode) et la stack réseau.

5 Implémentation des namespaces réseau dans ReactOS

L'un des points les plus importants est qu'il ne faut absolument pas casser la compatibilité des applications déjà existantes avec le système, en y incluant les namespaces. Tout doit être transparent. Pour ce qui est du code, pour rendre sa maintenance la plus facile possible, il est préférable de ne modifier qu'une petite partie du système et d'y centraliser toutes les modifications plutôt que de faire des changements plic-ploc un peu partout dans le kernel.

5.0.1 Gestion des Processus

Sous Linux, la création d'un namespace se fait via un flag dans le syscall `clone()`. L'apparition des différents namespaces sous Linux ont ajouté les flags suivant :

- **CLONE_NEWNS**
Isolation des points de montages
- **CLONE_NEWUTS**
Isolation du hostname et domain name
- **CLONE_NEWIPC**
Isolation des communications inter-processus
- **CLONE_NEWPID**
Isolation des PID, cela a pour effet de pouvoir avoir plusieurs processus avec le même PID et avoir différents `init` comme parent (PID 1)
- **CLONE_NEWNET**
Isolation de la couche réseau (routes, ARP, etc.)
- **CLONE_NEWUSER**
Isolation des user-id et group-id ce qui a pour effet de pouvoir séparer les permissions et les privilèges entre le namespace et l'extérieur (par exemple être root dans le namespace mais ne pas l'être dans l'environnement extérieur)

L'idée que j'ai retenue pour implémenter une isolation réseau sous ReactOS est de faire un procédé similaire. Sous Windows il n'existe pas de syscall POSIX `clone()` (ce syscall est utilisé par `fork()` et `pthread()`, par exemple). Pour lancer un nouveau processus, Windows dispose du syscall `CreateProcess()`. J'ai donc ajouté un flag que l'on peut passer en paramètre à `CreateProcess()`. Les flags existant sont (par exemple) :

- **CREATE_NEW_CONSOLE**

Le nouveau processus est ouvert dans une nouvelle console au lieu d'hériter de la console

- parente (par défaut)
- **CREATE_SUSPENDED**
Le thread principal est lancé en mode suspendu et ne sera actif que lors d'un appel à [ResumeThread\(\)](#)
- **DEBUG_PROCESS**
Active le mode de debug d'un processus (et de ses fils), ce qui permet l'utilisation de [WaitForDebugEvent\(\)](#) entre autres
- ...

Pour garder une cohérence entre Linux et Windows, j'ai fait un flag [PROCESS_CREATE_FLAGS_NEWNET](#). Ce flag permet de lancer une application (console par exemple, mais n'importe quelle application peut être exécutée par là. L'avantage de lancer une console est qu'on dispose alors d'un « launcher » pour lancer d'autres application dans le même namespace). Les namespaces sont automatiquement hérités de leurs parents, donc tout ce qui sera lancé depuis le namespace 1 (par exemple), à moins de n'avoir le flag qui crée un nouveau namespace spécifié, sera également attaché au namespace 1.

Tant qu'il existe au moins un processus dans un namespace, celui-ci est considéré comme actif. Une fois qu'il ne reste plus de processus attaché à un namespace, ce dernier est détruit et son ID sera le prochain utilisé.

5.0.2 Implémentation du gestionnaire de namespaces réseau

Pour ce qui est de l'implémentation, la création d'un namespace se fait dans [/ntoskrnl/ps/process.c](#) dans la fonction [PspCreateProcess\(\)](#). J'ai également créé un nouveau fichier dans [/ntoskrnl/netns/](#) qui se nomme [networknamespace.c](#) pour gérer les appels de drivers et la communication avec le kernel. Dans [/ntoskrnl/ps/kill.c](#) se trouve le code qui gère la fin d'un processus. Quand un processus est fermé (ou tué), le gestionnaire de namespace est appelé pour vérifier si il existe encore un processus dans le namespace dans lequel se trouvait le processus, si ce n'est pas le cas, le namespace est supprimé et ce qui y était lié est déchargé.

Pour stocker (faire le lien) entre un namespace et un processus, une bonne solution serait d'implémenter un système permettant d'isoler les processus entre eux, mais cette solution consiste plus à implémenter l'équivalent d'un [CLONE_NEWPID](#), ce qui n'est pas le but recherché. La méthode

la plus simple que j'ai trouvée a été d'attribuer un ID (champ `NetNs`) dans la table principale qui gère les processus, à savoir la table globale de structure `EPROCESS`. Cette table est la table utilisée par le système pour stocker le PID et tout le contexte d'utilisation d'un processus, ça me semble un bon endroit pour y stocker le namespace qui lui est lié, le seul souci qu'il peut y avoir avec cette solution est que la taille de la structure `EPROCESS` est modifiée. Cette table est utilisée entre autres par *WinDBG* (le débogueur Microsoft) et certains développeurs de ReactOS m'ont averti sur le channel IRC de `#reactos` que ça **pourrait** casser le fonctionnement de *WinDBG*. Je n'ai remarqué aucun problème avec celui-ci durant mon travail.

Dans mon implémentation, seul le namespace réseau est stocké en dur dans la table, pour étendre cette feature, remplacer ce champ par une structure pour y stocker l'ID des différents namespaces peut être une solution intéressante.

Pour ne pas devoir accéder à cette table depuis un driver/le kernel (tout d'abord parce que cette structure est opaque d'après la MSDN, et puis surtout par propreté et logique d'utilisation), j'ai fait une fonction qui permet de récupérer le numéro du namespace réseau courant via `IoGetCurrentProcessNs()`. Cet appel est utilisé un peu partout dans la lib ip et dans le protocol driver tcpip pour prendre une décision sur quelle table (routage, ARP, ...) utiliser.

La communication entre le kernel et la stack TCP/IP se fait via l'objet `\Device\NamespaceNetwork`. Cet objet est initialisé dans le driver `tcpip.sys` ce qui est loin d'être la meilleure solution, en effet cela implique que le gestionnaire de namespaces réseau doit attendre que `tcpip.sys` soit chargé pour pouvoir dispatcher les opérations du kernel. Dans le cas présent, c'était le moyen le plus facile de faire communiquer `tcpip.sys` et le kernel sans rien modifier d'autre dans le kernel.

Pour ne pas devoir créer un syscall par opération, la méthode la plus évidente est de faire un syscall qui prend une structure en paramètre et une opération tirée d'une liste. Côté kernel, la liste des opérations (opcode) que j'ai implémentées se trouve dans l'enum suivant :

Listing 1 – Extrait de /drivers/network/tcpip/include/dispatch.h

```
enum NETNSOPCODE {  
    KCREATE_NAMESPACE,  
    KREMOVE_NAMESPACE,  
    KATTACH_INTERFACE,  
    KDETACH_INTERFACE,  
    KSET_MACADDRESS,  
    KENABLE_SWITCH,  
    KDISABLE_SWITCH,  
};
```

Cette liste contient le minimum nécessaire pour créer et supprimer un namespace, mais en plus elle permet de gérer les « virtuals ethernet » (et les interfaces physiques également en réalité) car pour l’instant il n’y avait aucune commande implémentée pour attribuer une adresse IP ou MAC à une interface depuis la ligne de commande. Actuellement l’enum est copié/collé à plusieurs endroits dans le code car les dépendances d’inclues sont assez tordues pour pouvoir définir tout comme il faut partout. Il faut faire attention lors de la modification de l’enum de modifier les fichiers :

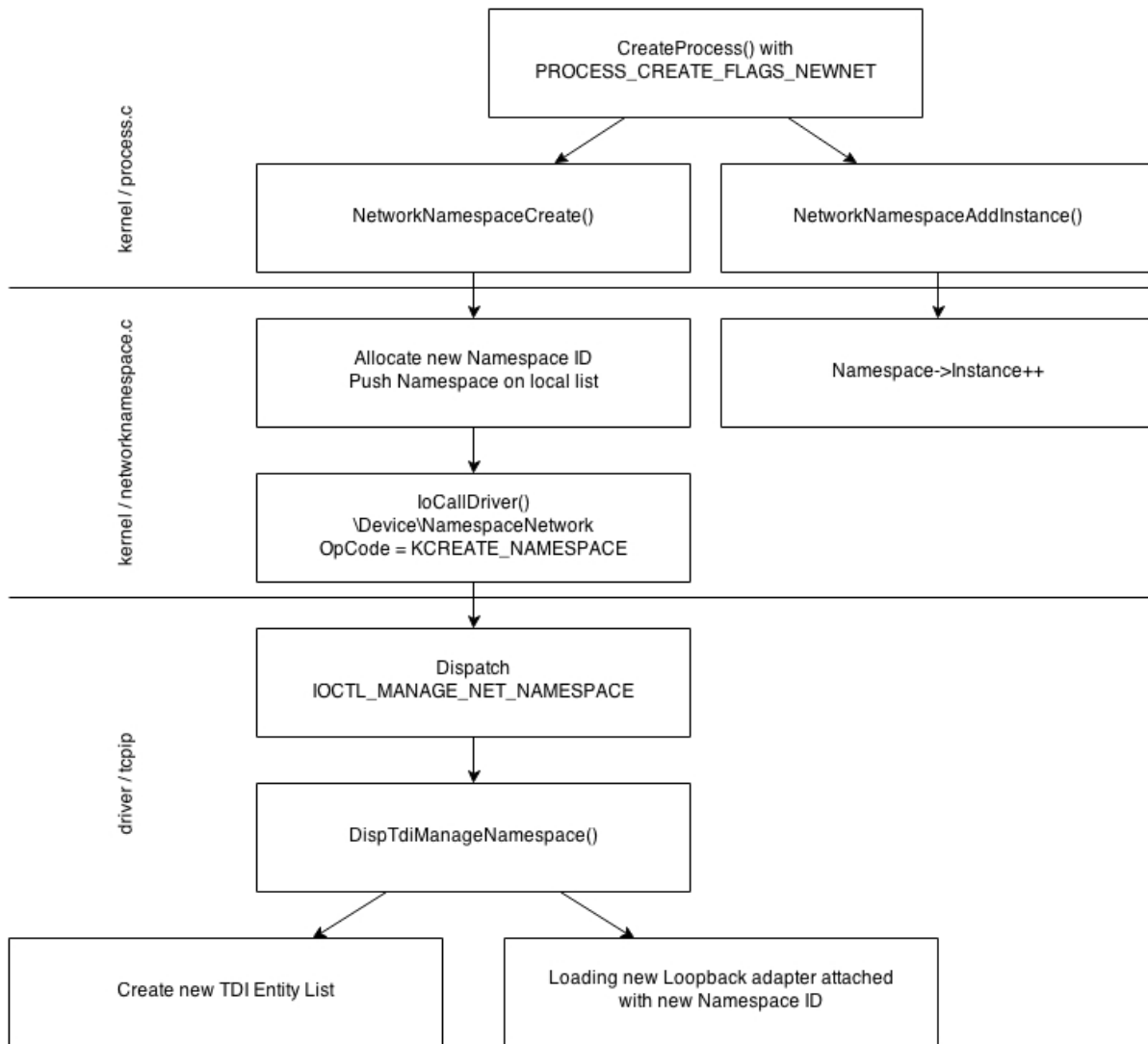
- /drivers/network/tcpip/include/dispatch.h
- /include/ndk/netns.h
- /include/psdk/netns.h
- /ntoskrnl/include/internal/networknamespace.h

Cet enum contient également la gestion du switch virtuel que j’ai implémenté pour gérer la couche 2 du modèle OSI (le switch est expliqué section 5.2.4 page 50).

Le syscall kernel (qui se trouve dans namespace.c) est `NtSetNetworkNamespace()`. Il prend en paramètre une structure `NETNSOP` qui contient un ID de namespace, un opcode (l’enum plus haut), une valeur (un `int` utilisé différemment en fonction de l’opcode) et une adresse MAC (qui n’est utilisée qu’avec l’opcode `KSET_MACADDRESS`).

Toutes les options sont préfixées par « K » pour différencier facilement la partie kernel-mode de la partie user-mode.

FIGURE 9 – Diagramme de création d'un namespace en kernel-space



(Notez que la partie « Create new TDI Entity List » sera expliquée plus loin).

Pour ce qui est de la partie usermode, les opcode sont définis :

Listing 2 – Extrait de /include/psdk/winbase.h

```

enum UNETNSOPCODE {
    ATTACH_INTERFACE,
    DETACH_INTERFACE,
    SET_MACADDRESS,
    ENABLE_SWITCH,
    DISABLE_SWITCH,
};
    
```

Comme on peut le voir, on ne peut pas créer de namespace via cet appel, il n'y a que via le `CreateProcess()` qu'on peut créer un nouveau namespace dans l'implémentation actuelle. Ces opcodes sont surtout faits pour interagir avec les namespaces existants. Les structures et les fonctions ont été créées, maintenant il reste à faire le lien entre le kernel et le usermode.

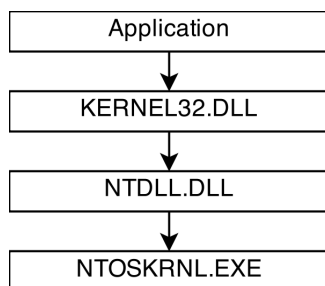
5.0.3 Ajout et linkage d'un syscall dans le kernel

Lors de la compilation, pour appeler le kernel depuis le usermode, il faut pouvoir compiler et linker l'exécutable correctement. Pour ça, il faut qu'à la compilation, le compilateur sache où se trouve l'appel système, comment l'appeler, etc.

Pour respecter la programmation Windows classique, pour faire un appel du user-mode au kernel-mode, il faut passer par la Win32 API. La Win32 API se trouve (en partie) dans `kernel32.dll` qui lui même appelle des fonctions de la Native API `ntdll.dll`.

`ntdll.dll` est la partie du système qui exporte les appels publics du kernel Windows (`ntoskrnl.exe`), appelés des « Native API ». Le code lié avec `ntdll.dll` est appelé « Native Application » et ne dépend pas de Win32 (ce code peut donc être appelé au début du chargement du système, avant que l'API utilisateur classique (Win32) soit utilisable, comme par exemple, `autochk.exe` qui s'occupe de lancer `CheckDisk` au démarrage). `Kernel32.dll` est donc une « Native Application » qui exporte au reste du système des appels de plus haut niveau (gestion de threads, mémoire, etc.).

FIGURE 10 – Diagramme de la couche d'appel du kernel à la Win32 API



Concrètement, dans ReactOS, `ntdll.dll` et `kernel32.dll` ont dans leurs dossiers sources, un fichier `kernel32.spec` et `ntdll.spec` qui contient la liste des fonctions à exporter ainsi que la taille et le nombre de paramètres que la fonction prend.

Listing 3 – Code ajouté à ntdll.spec, ntoskrnl.spec et kernel32.spec

```
@ stdcall NtSetNetworkNamespace(ptr) ; ntdll.spec
@ stdcall IoGetCurrentProcessNs() ; ntoskrnl.spec
@ stdcall SetNetworkNamespace(ptr) ; kernel32.spec
```

Il ne reste plus qu'à ajouter les prototypes dans un `.h` global (comme `winbase.h` par exemple), et le tour est joué. Maintenant que le syscall est fait et exporté, il reste à l'utiliser. Tout ces appels ont été utilisés en user-mode avec l'utilitaire en ligne de commande « `unshare.exe` » que j'ai spécialement développé dans le cadre de ce travail.

5.0.4 Unshare

Sous Linux, l'application « `unshare` » permet de lancer une application dans un nouveau namespace (au choix). Pour l'exemple ci-dessous, on va lancer un `bash` dans un nouveau namespace réseau et on va voir que la liste des interfaces disponibles sur le système ne sera pas la même qu'avant.

Listing 4 – Utilisation de « `unshare` » sous Linux

```
# unshare --help
-m, --mount          unshare mounts namespace
-u, --uts            unshare UTS namespace (hostname etc)
-i, --ipc           unshare System V IPC namespace
-n, --net           unshare network namespace
-p, --pid           unshare pid namespace
-U, --user          unshare user namespace
-f, --fork          fork before launching <program>
  --mount-proc[=<dir>] mount proc filesystem first (implies --mount)

# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode [...]
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP [...]
   link/ether 8c:89:a5:c8:8a:84 brd ff:ff:ff:ff:ff:ff
```

Listing 5 – Nouveau namespace avec « unshare » sous Linux

```
# unshare --net bash
# ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Le nouveau namespace réseau est volatile (le namespace est détruit une fois que bash est quitté). J'ai voulu reprendre cette idée de pouvoir créer un namespace via une commande `unshare`, mais en plus de ça, cette commande est devenue la commande principale de gestion des namespaces, des ip, des adresses mac, etc. Par facilité, l'implémentation de `unshare.exe` est clairement un « fourre-tout », l'idée est de supporter pleinement l'aspect kernel de l'isolation, le nom du programme qui va changer l'adresse IP ou l'adresse MAC de l'interface a peu d'importance vis-à-vis de l'implémentation dans le système. Pour plus de précisions sur les utilitaires qui devraient être utilisés, voir section 7.2 page 68.

Pour ce qui est de la partie volatile, j'ai gardé ce système pour le portage. Cela facilite le développement sans pour autant enlever des fonctionnalités. Faire du code persistant nécessiterait de sauver les ID des namespaces et les interfaces qui y sont attachées. Pour garder une persistance, à l'heure actuelle, il faut créer un script de démarrage qui lance à la manière d'un batch, toutes les commandes `unshare.exe` nécessaires, un peu à la façon d'`iptables` sous Linux.

Le code d'`unshare.exe` se trouve dans `/base/applications/network/unshare/` et sa syntaxe est relativement simple, on la verra au cours du développement qui suit ⁴.

Maintenant que les processus peuvent utiliser différents namespaces, voyons comment nous allons exploiter ça dans le driver `tcpip.sys` et comment nous allons trainer les I/O réseaux pour avoir un fonctionnement cohérent et isolé.

5.1 Implémentation des namespaces dans la stack réseau (`tcpip.sys`)

Le fichier `tcpip.sys` s'occupe entièrement du traitement réseau dans le système. Même si le résultat final est un gros fichier binaire qui contient tout, le code, lui, est bien segmenté suivant les différents traitements (couche OSI 2, 3 et 4).

4. Un résumé de la syntaxe se trouve en annexe, page 79

Le principe à suivre est simple. Côté usermode, on demande quelque chose au kernel (à un driver) et le système nous donne une réponse. Cette réponse est généralement une liste d'éléments (liste d'interfaces, routes, adresses, ...). Il « suffit » donc de trouver où cette liste est générée et d'y ajouter un matching sur le namespace en plus que le traitement qui est actuellement fait. L'idée est là. Pour ce faire, on va analyser le code qui s'occupe du layer 3.

5.1.1 Bibliothèque IP

Comme dit plus haut, la partie TCP/IP du système utilise 2 grandes parties, le code de `tcpip.sys` et la bibliothèque `ip` qui est statiquement liée.

La lib `ip` qui se trouve dans `/lib/drivers/ip/` est structurée ainsi (je n'ai gardé que les fichiers qui ont été utilisés dans ce travail) :

- `/lib/drivers/ip/network` :
 - `address.c` : permet de stocker une adresse ip (ipv4 et ipv6) et de manipuler, tester et les comparer. Il y a également des fonctions de debug intéressantes dedans dont une qui permet de formater une adresse pour pouvoir l'afficher en tant que string.
 - `arp.c` : son nom peut être trompeur, il s'agit du handler de paquets ARP et non de la gestion de la table ARP du système. Les fonctions contenues dans ce fichier s'occupent soit de forger une requête ARP et de l'envoyer sur le réseau, soit de lire un paquet ARP et d'en analyser le contenu et traiter (ou pas) la requête.
 - `icmp.c` : son nom l'indique clairement, ce fichier contient les fonctions pour envoyer ou recevoir un paquet ICMP.
 - `interface.c` : permet de manipuler l'état et des informations sur une interface ip.
 - `ip.c` : la partie principale de la bibliothèque, ce fichier contient ce qu'il faut pour initialiser la bibliothèque, enregistrer des protocoles, des interfaces et communiquer avec `tcpip.sys`
 - `loopback.c` : tout le code relatif aux interfaces de loopback. L'idée que j'avais du loopback sous Windows était qu'une interface NDIS (comme tout le reste du système) s'occupait de gérer ça, mais en fait non, c'est une interface virtuelle de niveau 3 qui s'occupe de tout gérer (en tout cas dans ReactOS).
 - `neighbor.c` : la gestion des voisins (les adresses MAC des voisins, donc la table ARP en définitive).
 - `receive.c` : dispatcheur de paquets reçus.
 - `router.c` : s'occupe de la gestion (ajout, suppression, recherche) de route dans la table de routage.

— `/lib/drivers/ip/transport` :

contient des sous-dossiers (`tcp`, `udp`, `rawip`) contenant la gestion des différents protocoles de couche 4. Toute l'isolation étant faite après la couche 3, rien de cette partie n'a été modifié, il n'y avait aucune raison d'y toucher.

Bien que ça soit une bibliothèque (en tout cas d'après le path du code), une chose étonnante est que tout les includes se trouvent dans le dossier de `tcpip` (dans `/drivers/`) et non dans la lib avec le reste. On voit donc très clairement qu'il y a une dépendance mutuelle entre les deux parties.

5.1.2 Adresses

Les adresses IP sont stockées sous forme d'une structure pouvant aussi bien accepter une IPv4 qu'une IPv6.

Listing 6 – struct `IP_ADDRESS`

```
typedef struct IP_ADDRESS {
    UCHAR Type ;
    union {
        IPv4_RAW_ADDRESS IPv4Address ;
        IPv6_RAW_ADDRESS IPv6Address ;
    } Address ;
} IP_ADDRESS, *PIP_ADDRESS ;
```

Le `Type` définit si il s'agit d'une IPv4 ou une IPv6 (`0x04` ou `0x06`, défini par les deux defines `IP_ADDRESS_V4` et `IP_ADDRESS_V6`) puis l'union contient l'adresse en question.

5.1.3 Interfaces

Les adresses IP ne sont pas attribuées à des interfaces NDIS ou des interfaces layer 2 directement. La bibliothèque contient ses propres interfaces (liées à l'interface de la couche inférieure par un pointeur) et crée donc une indépendance à ses couches inférieures. Une interface IP est définie comme telle (seul les champs intéressants pour ce travail ont été gardés) :

Listing 7 – struct IP_INTERFACE

```
typedef struct _IP_INTERFACE {  
    // [...]  
    LIST_ENTRY ListEntry;  
    PVOID Context;  
    IP_ADDRESS Unicast;  
    IP_ADDRESS Netmask;  
    IP_ADDRESS Broadcast;  
    UNICODE_STRING Name;  
    UINT Index;  
    LL_TRANSMIT_ROUTINE Transmit;  
    // [...]  
} IP_INTERFACE, *PIP_INTERFACE;
```

Les noms de variables sont globalement clairs, mais quelques précisions :

- **ListEntry** : champ utilisé par les listes chaînées implémentées dans le code
- **Context** : un pointeur void vers quelque chose d'intéressant de niveau inférieur. Dans notre cas, ce pointeur sera le lien vers l'interface layer 2 à laquelle l'interface IP est attachée.
- **Index** : numéro unique d'identification de l'interface. C'est cet index qu'on retrouvera comme numéro d'interface lors d'un route print par exemple. Pour communiquer avec le usermode, c'est cet index qui sera utilisé pour spécifier une interface, c'est plus facile que de matcher des noms.
- **Transmit** : pointeur de fonction vers un handler qui va dispatcher un paquet pour l'envoyer vers la couche inférieure.

Cette structure a été adaptée en fonction des différents besoins qui sont expliqués dans ce chapitre.

5.1.4 IP

Le point d'entrée et le coeur de la bibliothèque IP. Bien que cette partie n'ait presque pas été modifiée, il est important de comprendre son fonctionnement pour savoir comment tout est lié ensemble.

C'est dans la routine d'initialisation du driver TCPIP que l'initialisation de la bibliothèque est appelée. Elle met en place ce qu'il faut pour gérer les listes d'interface et prépare l'enregistrement des protocoles.

Les protocoles ICMP, TCP et UDP s'enregistrent comme handlers et lors de la réception d'un paquet, une itération sur les handlers s'effectue pour savoir à qui envoyer le paquet pour le traiter. De cette façon il est très facile d'implémenter un nouveau protocole car il s'agit de créer un nouveau traitement et l'enregistrer comme les autres sans modifier le code déjà existant. L'avantage pour moi, c'est qu'il suffit d'adapter le coeur pour que tout ce qui s'y attache puisse bénéficier de l'isolation de manière transparente, c'est pour ça que la couche 4 n'a pas à être modifiée.

C'est également dans cette partie de code qu'on ajoute une interface IP. Pour ce faire, il suffit de remplir une structure `PLLIP_BIND_INFO` et de la passer à la fonction `IPCreateInterface` qui va s'occuper d'ajouter une nouvelle interface à la liste globale. La structure `PLLIP_BIND_INFO` est en fait une `IP_INTERFACE` très simplifiée qui ne contient que le minimum pour la lier au reste du système : le contexte de la couche inférieure, la taille du header de la couche inférieure (pour savoir ce qu'il faut ignorer lors de la réception d'un paquet) et un pointeur vers une fonction qui permet de préparer un paquet à envoyer.

5.1.5 Loopback

La partie loopback a été un point maître dans l'implémentation car c'est la façon la plus simple de pouvoir tester l'isolation d'interface. Sous Linux, quand on crée un nouveau namespace, automatiquement une interface de loopback (à l'état down) est ajoutée à ce namespace. Linux permet une utilisation du réseau sans interface de loopback ce qui n'est pas le cas sous Windows. En effet, pour ce qui est de ReactOS, toutes les manipulations de recherche d'interface, etc., partent du principe qu'il y a **TOUJOURS** une interface existante, la loopback. Cela simplifie le code et le principe, étant donné qu'il ne faut pas prendre en compte le cas où aucune interface n'existe. Le revers de la médaille est qu'on ne donne pas un accès complet à la couche réseau à l'utilisateur. Il est (sans hack du kernel) impossible de changer l'adresse ip de l'interface de loopback ni même de désactiver l'interface.

En partant de ce principe, leur idée (que je trouve justifiée) a été de mettre une variable globale dans la bibliothèque qui contient les informations à propos de la loopback (un pointeur vers sa structure `IP_INTERFACE` en d'autres termes). C'est là que ça se corse. L'implémentation n'est **ABSOLUMENT** pas prévue pour supporter plusieurs interfaces de loopback. Pour avoir une isolation correcte des namespaces, il est normal d'avoir des loopbacks indépendantes qui peuvent toutes avoir la même adresse IP (typiquement 127.0.0.1/8). Il a donc fallu trouver un moyen de

gérer plusieurs loopbacks (si possible sans réécrire tout ce qui existe déjà...)

La méthode qui me semble la plus logique est de commencer par remplacer la variable globale par une liste de loopbacks et non par une loopback unique. En utilisant le système de liste chaînée du système, pas besoin de faire grand chose :

Listing 8 – struct LOOPLIST

```
typedef struct _LOOPLIST {  
    LIST_ENTRY ListEntry;  
    PIP_INTERFACE Loopback;  
} LOOPLIST, *PLOOPLIST;
```

Cette méthode sert à isoler la liste des loopbacks indépendamment du reste du code, mais plus loin, on se rend compte que pour savoir si une interface précise est une loopback, ils comparent l'adresse de la variable globale à l'interface. Cette méthode (un peu trop naïve) est adaptable en remplaçant la comparaison par un appel de fonction qui itère la liste des loopbacks pour savoir si elle est présente dans la liste ou pas. Cette méthode n'est pas idéale quand on peut faire beaucoup plus simple : ajouter un flag à la structure `IP_INTERFACE` pour savoir si c'est une loopback ou pas. De cette façon, on sait directement si une interface précise est une loopback. C'est beaucoup plus facile et efficace qu'une liste. A noter que j'ai quand même laissé la liste des loopbacks dans le code, ça laisse une abstraction de dépendances (liste de loopbacks, liste d'interfaces).

Lors de la création d'une loopback, elle s'identifie comme étant dans le namespace 0 par défaut. C'est au code qui a appelé la création de la loopback de changer son namespace pour la mettre dans le bon lui-même.

5.1.6 Voisinage

Le voisinage (la table ARP en d'autres termes) est une table temporaire (les champs qui y sont stockés, y sont en cache avec un timeout, une fois le timeout dépassé, l'entrée est supprimée) qui fait la correspondance entre un adresse IP et une adresse MAC (qui a été résolue par le protocole ARP). Cette table est nécessaire pour envoyer une trame ethernet vu qu'on doit placer l'adresse MAC de destination dans le header ethernet.

La table de voisinage est faite comme suit :

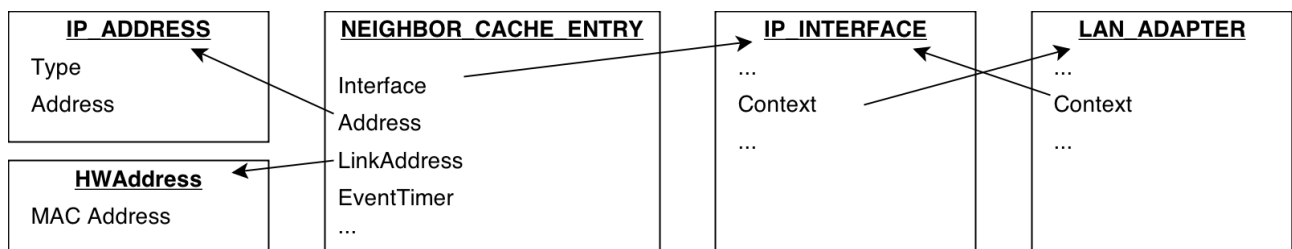
Listing 9 – struct NEIGHBOR_CACHE_ENTRY

```
typedef struct NEIGHBOR_CACHE_ENTRY {  
    // [...]  
    UCHAR State;  
    UINT EventTimer;  
    UINT EventCount;  
    PIP_INTERFACE Interface;  
    PVOID LinkAddress;  
    IP_ADDRESS Address;  
    // [...]  
} NEIGHBOR_CACHE_ENTRY, *PNEIGHBOR_CACHE_ENTRY;
```

- **State** : état de la résolution (reachable, stale, delay, incomplete, failed, ...) ⁵
- **EventTimer** : durée (ticks) depuis le dernier événement
- **EventCount** : nombre d'événements
- **Interface** : interface depuis laquelle l'entrée ARP a été ajoutée (donc l'interface par laquelle il faut sortir un paquet vers cette destination)
- **LinkAddress** : pointeur vers l'adresse de layer 2 (dans notre cas, une adresse MAC)
- **Address** : adresse IP de destination

Pour avoir une idée de comment c'est représenté en mémoire, voilà un diagramme des éléments accessibles depuis une entrée dans la table ARP du système :

FIGURE 11 – Diagramme de relation avec une entrée dans la table ARP



5. Voir le fonctionnement du protocole ARP, bon exemple : <http://linux-ip.net/html/ether-ARP.html>

5.1.7 Routeur

Pour ce qui est du routage, le système garde une liste de destination (la FIB pour Forwarding Information Base). Cette table contient :

Listing 10 – struct FIB_ENTRY

```
typedef struct _FIB_ENTRY {
    LIST_ENTRY ListEntry;
    // [...]
    IP_ADDRESS NetworkAddress;
    IP_ADDRESS Netmask;
    PNEIGHBOR_CACHE_ENTRY Router;
    UINT Metric;
} FIB_ENTRY, *PFIB_ENTRY;
```

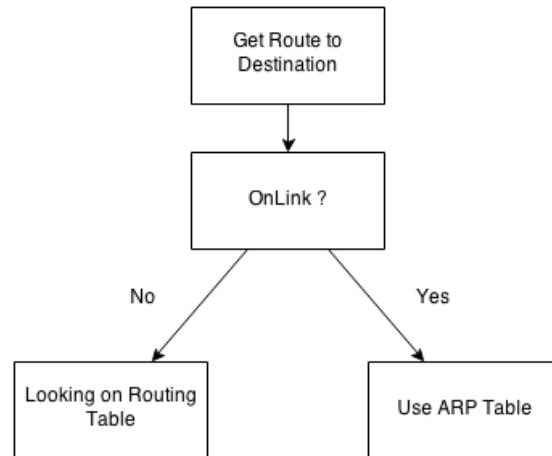
On y a toutes les informations nécessaires : l'adresse de destination avec son masque, l'entrée dans la cache ARP pour y accéder et son metric. L'entrée dans la table ARP contient un pointeur vers l'interface de sortie. Une fois qu'on a accès à l'interface on a donc accès à son namespace. Tout ce qu'il nous faut donc. Il reste à modifier le code pour qu'une recherche de route tienne compte du namespace en cours (ou un namespace voulu).

La fonction `RouterGetRoute()` s'occupe de rechercher dans la table de routage, une route vers une destination donnée. Hélas, on ne peut pas juste s'occuper de comparer les namespaces puis renvoyer ce qu'on a trouvé. Durant le développement, j'ai découvert que lors de l'émission d'un paquet, tout se passe bien (le namespace étant donné par le processus en cours, ça correspond), mais lors de la réception d'un paquet, le processus en cours d'exécution est `SYSTEM` (pid 4 et de même sous Windows). Du coup, tous les processus de base étant par défaut dans le namespace 0, la réception d'un paquet ne se routait pas correctement et le paquet se retrouvait drop. Il a donc fallu vérifier si c'est le processus `SYSTEM` qui fait la demande ou pas. Si oui, le système a une vision globale de la table et non pas uniquement son namespace.

Pour comparer le processus `SYSTEM`, j'ai hard-codé le numéro du PID `0x04`, ce n'est pas idéale mais je ne sais pas trop comment faire ça de façon plus générique. J'ai vu dans la doc MSDN qu'il y avait une routine qui s'appelle `PsIsSystemThread()` qui pourrait résoudre le problème, mais je n'ai pas eu l'occasion de la tester.

Lors de la recherche d'une route, il y a un cas particulier qui est traité, c'est le cas où l'adresse de destination se trouve sur une interface locale. Lors d'une recherche de route vers une destination, une première étape est de voir si il existe une interface qui contient la destination, si oui on utilise la cache ARP sinon on regarde dans la table de routage classique.

FIGURE 12 – Diagramme de recherche d'une route



Lors de la recherche OnLink, j'ai dû adapter un peu la recherche en faisant une nouvelle fonction qui prend, en plus, comme argument l'interface source de la requête. En fait, lors d'une communication inter-namespace (qui sera expliquée plus en détail dans la partie Switching (section 5.2.4 page 50)), la requête source se fait depuis le PID `0x04` aussi, alors que la source ne vient pas du monde extérieur mais bien du système. Il est donc possible de connaître la source et donc de pouvoir préciser la recherche à faire. En effet, sinon c'est sur toutes les routes et interfaces que la recherche se fait et ça pose problème en cas d'overlapping : quelle interface choisir vu qu'elles ont la même ip ? En se basant sur l'interface source, il est possible de faire correspondre les namespaces en sachant lesquels sont inter-connectés (via le contexte Layer 2, expliqué également plus loin).

La couche 3 (le fonctionnement et l'utilisation du protocole IP) est maintenant adaptée pour pouvoir gérer du trafic depuis des namespaces isolés.

Concrètement, on a modifié le fonctionnement de la table de routage et adapté les interfaces de loopback.

Lors du développement, après être arrivé à faire de l'isolation de ping entre 2 loopbacks dans

2 namespaces différents (les loopbacks n'ont pas d'adresse MAC, il s'agit donc bien de couche 3 uniquement), une grande étape a été franchie. Cependant, isoler en couche 3 n'est pas suffisant, en effet pour l'instant, on est capable d'avoir du routage IP isolé, mais le broadcast ethernet ne l'est pas du tout et le protocole ARP (par exemple) utilise du broadcast ethernet pour pouvoir résoudre les adresses MAC en fonction des adresses IP. Sans isolation de couche 2, cette communication ne peut se faire correctement.

Voyons voir comment traiter la couche 2.

5.2 Implémentation layer 2 et connexion au layer 3

Pour poursuivre le processus d'isolation, il faut pouvoir différencier les différents LAN (des Virtuals LAN) qui sont utilisés. Un namespace correspond à un LAN (avec un domaine de broadcast qui lui est propre). Pour pouvoir isoler un LAN, il faut pouvoir isoler et dispatcher une trame ethernet vers le bon endroit (un « VLAN 1 » ne doit pas recevoir des trames de broadcast du « VLAN 2 » par exemple).

Pour y parvenir, on va attaquer une partie plus bas niveau au niveau driver, car on ne va plus travailler uniquement sur une couche software type bibliothèque de gestion de paquets, mais bien d'un driver qui va communiquer avec des cartes réseaux, des adresses MAC, etc. (bien entendu, la partie hardware n'entre pas en jeu, tout a été écrit pour ne pas nécessiter de modification extérieure... si il avait fallu réécrire les drivers des cartes réseaux pour implémenter les namespaces, l'intérêt serait bien moindre).

Comme expliqué lors du début de cet ouvrage, Windows et ReactOS utilisent NDIS pour communiquer avec le matériel réseau. Les drivers Miniport NDIS s'occupent de la partie hardware et font le lien avec la partie NDIS ProtocolDriver (dans notre cas, TCP/IP).

Le code du ProtocolDriver TCP/IP se trouve dans le fichier `/drivers/network/lan/lan/lan.c` et son point d'entrée est la fonction `LANRegisterProtocol()`. Pendant le chargement du driver `tcpip.sys`, cette fonction est appelée et le driver se lie (bind) avec NDIS. Travailler avec NDIS, globalement, c'est appeler des `XxxxRegister` avec comme paramètre une structure de données qui ne contient que des pointeurs de fonctions avec des prototypes bien précis, qui sont décrits dans une documentation officielle⁶. C'est le cas que ça soit du Miniport ou des ProtocolDriver. Une

6. <http://www.ndis.com/>

fois le ProtocolDriver correctement attaché, NDIS va appeler le callback `LANRegisterAdapter()` pour chaque interface (Miniport) existante (si une nouvelle interface Plug'n'Play vient à arriver pendant l'exécution du système, cette fonction sera également appelée avec la nouvelle interface).

Pour représenter une interface layer 2, au même titre que pour la partie IP, une structure propre aux interfaces layer 2 est utilisée, nommée `LAN_ADAPTER` :

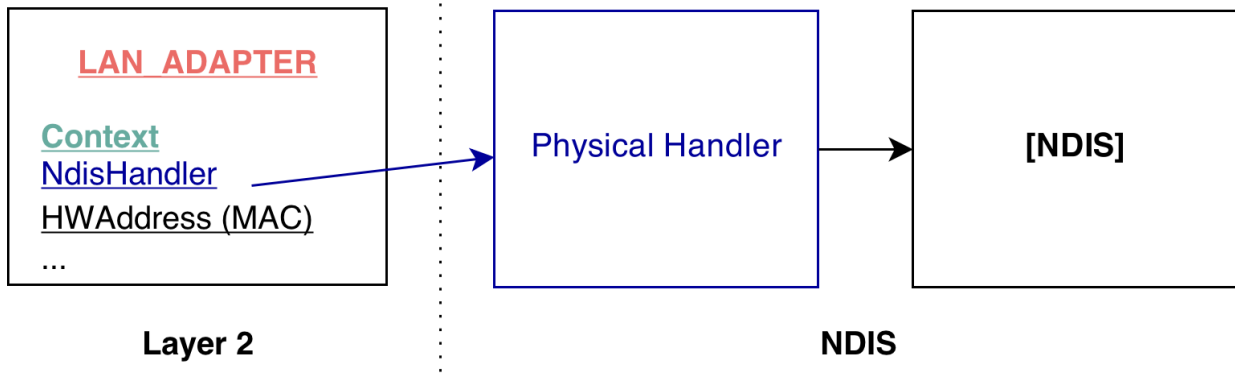
Listing 11 – struct `LAN_ADAPTER`

```
typedef struct LAN_ADAPTER {
    LIST_ENTRY ListEntry;
    // [...]
    PVOID Context;
    NDIS_HANDLE NdisHandle;
    NDIS_STATUS NdisStatus;
    NDIS_MEDIUM Media;
    UCHAR HWAddress[IEEE_802_ADDR_LENGTH];
    // [...]
    UCHAR HeaderSize;
    USHORT MTU;
    UINT Speed;
    UINT PacketFilter;
    // [...]
} LAN_ADAPTER *PLAN_ADAPTER;
```

Encore une fois, seuls les champs qui sont utilisés dans le cadre de ce développement ont été notés. Pour ce qui est des précisions :

- **Context** : pointeur vers la structure `IP_INTERFACE` qui lui est attachée (voir section 5.2.3 page 48 pour avoir un diagramme plus clair).
- **NdisHandle** : un identificateur unique utilisé par NDIS pour savoir où dispatcher les demandes à propos de l'interface
- **Media** : contient un ID sur le type d'interface dont il s'agit. Seul le média ethernet (802.3) est supporté dans ReactOS.
- **HWAddress** : sans doute le champ le plus important et le plus intéressant : l'adresse MAC de l'interface
- **PacketFilter** : flags NDIS pour filtrer les paquets qu'on va recevoir.

FIGURE 13 – Diagramme de lien entre LAN_ADAPTER et NDIS



Petite parenthèse : étant donné que plus loin on va devoir recevoir des trames dont l'adresse MAC sera l'adresse d'une interface virtuelle, il va falloir activer le mode promiscuous de la carte réseau pour ne pas que la carte (de façon hardware) drop les trames qui ne lui sont pas directement adressées. Pour cela, il suffit d'ajouter le flag `NDIS_PACKET_TYPE_PROMISCUOUS` au filtre de paquet de l'interface (PacketFilter).⁷

Lorsque l'interface Miniport reçoit une trame, elle la forward à NDIS qui s'occupe de forwarder le paquet aux ProtocolDrivers. Le `ReceiveHandler` est alors appelé avec la trame et l'interface source en paramètre.

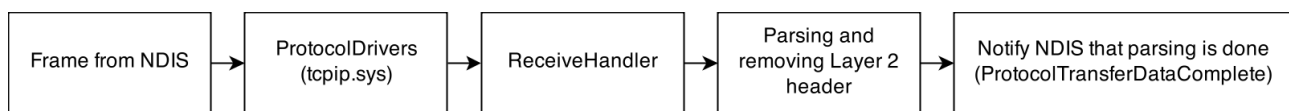
5.2.1 Réception d'une trame dans le driver TCP/IP

La réception d'une trame est plutôt basique, le driver se charge simplement de regarder le type de data dans le payload puis de retirer le header ethernet et forwarder le paquet au handler de paquets.

Seul les types IPv4, IPv6 et ARP sont implémentés dans ReactOS.

Une fois cette opération terminée, le driver notifie NDIS en disant que le traitement de trame est terminé et demande de forwarder le paquet résultant au handler de paquets (`ReceivePacketHandler`).

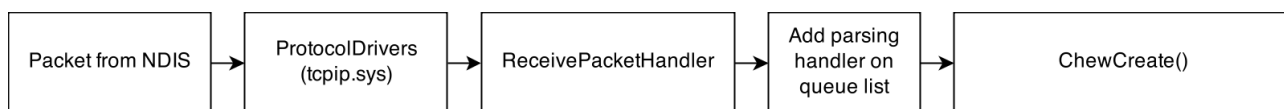
FIGURE 14 – Diagramme de réception d'un paquet depuis NDIS



⁷. Le driver ethernet *pcnet* qui a été implémenté dans ReactOS ne semble pas avoir de handler permettant d'activer ou pas le mode promiscuous. Voir section 11.6.1 page 81

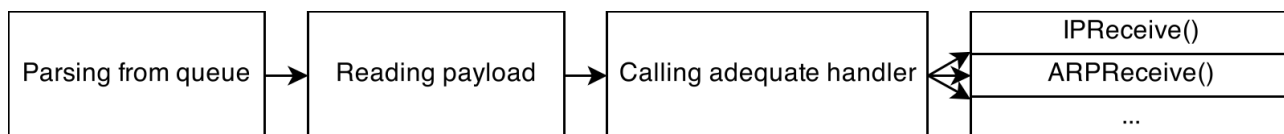
Une fois que le handler de paquets est appelé, le traitement du paquet est ajouté dans une queue de traitements. Cette queue est gérée par une implémentation de « Chew » ([ChewCreate](#)), qui prend en paramètres une fonction et un paramètre (dans ce cas ci, une structure qui contient tout ce qui est nécessaire : paquet, interface, etc.).

FIGURE 15 – Diagramme de préparation d’un paquet NDIS reçu



Une fois que la queue arrive au paquet en question, le contenu est analysé et extrait du paquet NDIS (qui utilise un format propre au protocole pour stocker les données dans leur structure, ce n’est pas un simple buffer). Les statistiques de l’interface sont mises à jour (nombre de paquets/bytes reçus) puis selon le type de payload, le handler correspondant est appelé ([IPReceive](#) ou [ARPReceive](#), le reste est droppé). C’est à partir de ce moment là que la bibliothèque IP prend le relais.

FIGURE 16 – Diagramme de réception d’un paquet depuis NDIS vers TCP/IP



Ce ProtocolDriver est le premier à avoir un paquet entrant après NDIS, il est du coup également le dernier à recevoir un paquet sortant.

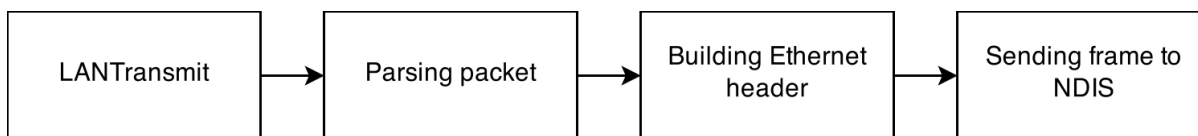
5.2.2 Emission d’une trame depuis le driver TCP/IP

Chaque interface IP (donc de couche 3) dispose d’un pointeur de fonction vers un callback de transmission. Excepté pour la loopback dont le code de transmission se trouve dans la bibliothèque IP (il n’a pas de passage via NDIS pour la loopback dans ReactOS), la fonction de transmission est [LANTransmit](#) qui se trouve dans le ProtocolDriver TCP/IP.

Une fois que cette fonction est appelée (elle dispose du paquet et de l’interface source qui veut

émettre), elle se contente de créer une nouvelle trame ethernet, d'y placer le paquet (IP) en payload et de construire l'en-tête ethernet (MAC source et MAC destination) en fonction des paramètres fournis. Elle met également à jour les statistiques de l'interface (nombre de paquets/bytes émis) puis forward la trame nouvellement créée à NDIS.

FIGURE 17 – Diagramme d'émission d'un paquet depuis TCP/IP vers NDIS



Jusque là, il s'agit du fonctionnement d'origine du driver TCP/IP et aucune modification n'y a été vraiment apportée pour y supporter les namespaces... mais il faut savoir comment le protocole fonctionne avant de le triturer pour y inclure quelque chose de nouveau. L'étape suivante, qui est l'étape clé dans les containers (et donc les namespaces), est l'utilisation d'interfaces virtuelles pour faire un pont entre l'intérieur isolé du namespace et l'extérieur.

5.2.3 Virtual Ethernet

L'interface virtuelle que j'ai implémentée est une interface de couche 3 qui se trouve dans la bibliothèque IP et qui se base sur l'interface de loopback (étant donné que la loopback est également une interface virtuelle où l'émission et la réception de paquet sont communes). Cependant, elle nécessite également d'être liée à la couche 2 pour disposer d'une adresse MAC.

Son implémentation se trouve dans [/lib/drivers/ip/network/veth.c](#) et va toucher aussi bien à la couche 3 que la couche 2.

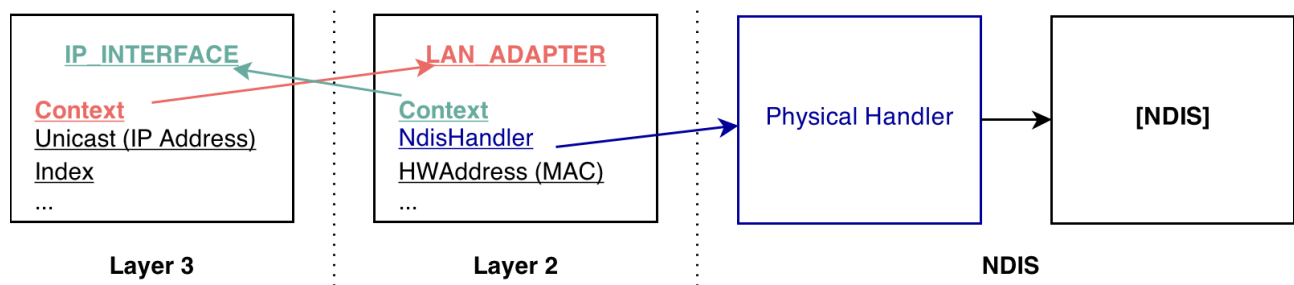
Je pars du principe qu'une interface virtuelle, dans le cas dont j'ai besoin, doit être liée à une interface déjà existante (sinon ça revient au même que de faire une loopback). La première étape est donc d'identifier l'interface demandée. Vu qu'on ne dispose que de listes linéaires, on parcourt la liste des interfaces existantes et on compare l'index de l'interface avec l'index passé en paramètre (typiquement, l'index donné lors de la commande `unshare attach <ID>`).

Ensuite, tout comme une loopback, on demande à la bibliothèque IP d'instancier une `IP_INTERFACE` qu'on va ensuite configurer.

Le but d'une virtual ethernet est de s'attacher à une interface physique⁸ (ou plutôt de manière générale à un NDIS Miniport). Cela veut dire que pour communiquer avec NDIS, il nous faut récupérer le NdisHandler de l'interface à laquelle on veut se connecter. En fait pour faire encore plus simple, on va tout simplement dupliquer la structure `LAN_ADAPTER` de l'interface à laquelle on s'attache, et s'y attacher.

L'avantage de ce cas est qu'on dispose exactement du même traitement que l'interface source (émission, réception de paquets, etc.), et ayant dupliqué la structure, on peut également la modifier... donc changer l'adresse MAC de l'interface ! Allons-y. Une fois la structure dupliquée, on génère une adresse MAC aléatoire (voir section 7.2 page 67) puis on lie la nouvelle `LAN_ADAPTER` au contexte de notre nouvelle `IP_INTERFACE`.

FIGURE 18 – Diagramme d'un lien classique entre `IP_INTERFACE`, `LAN_ADAPTER` et NDIS

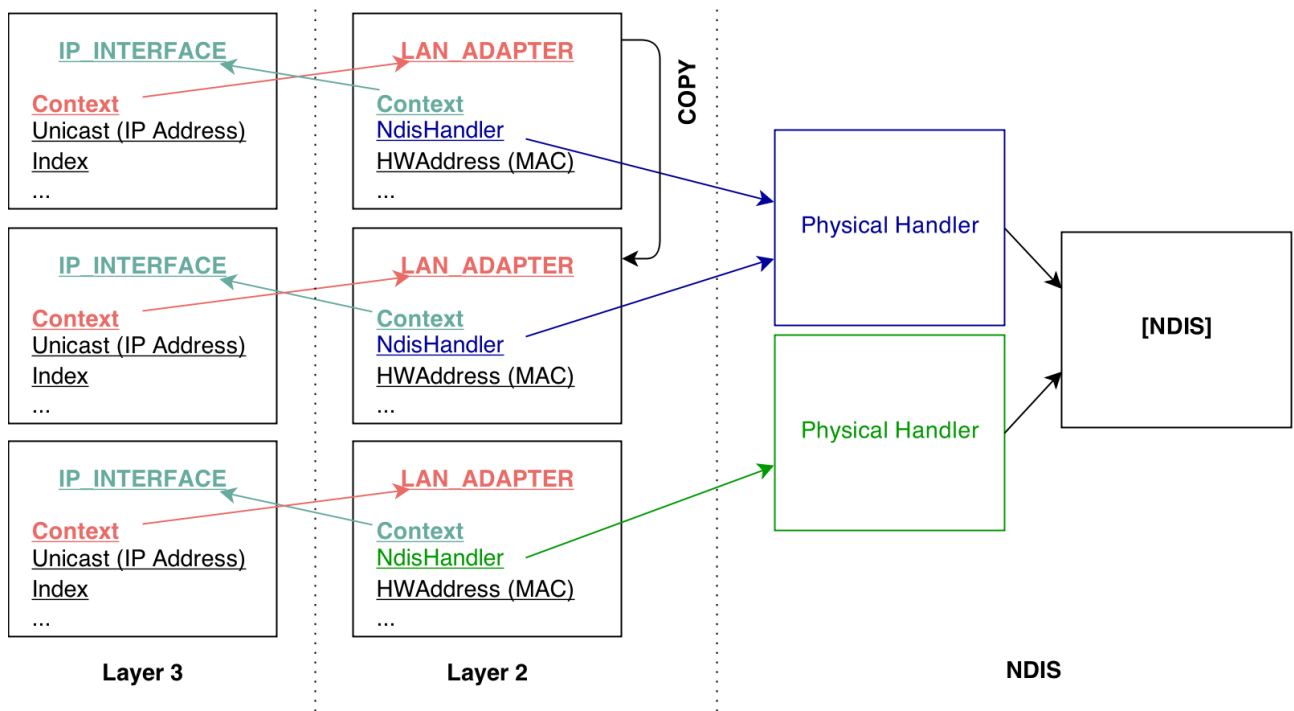


L'interface virtuelle étant attachée à une interface physique, ça veut dire que d'un point de vue réception de paquets, l'interface source qu'on va recevoir sera l'interface physique et non l'interface virtuelle (vu que NDIS ne connaît pas l'existence de **nos** interfaces virtuelles). Il va donc falloir ajouter une couche entre la réception du paquet et son traitement, une couche qui va consister à aiguiller le paquet vers la bonne direction⁹.

8. Rien n'empêche le code de s'attacher à une loopback ou même à une autre virtual ethernet, je ne vois pas dans quel cas cela ne serait pas possible, cependant le code n'a pas été testé pour, cette utilisation serait donc expérimentale.

9. Attention, la Virtual Ethernet qui a été créée ici n'est pas réellement comparable à une « veth » sous Linux. Sous Linux, ce type d'interface fonctionne comme une pipe (tout ce qui entre d'un côté, sort de l'autre), dans notre cas, une Virtual Ethernet nécessite un switch virtuel pour fonctionner correctement

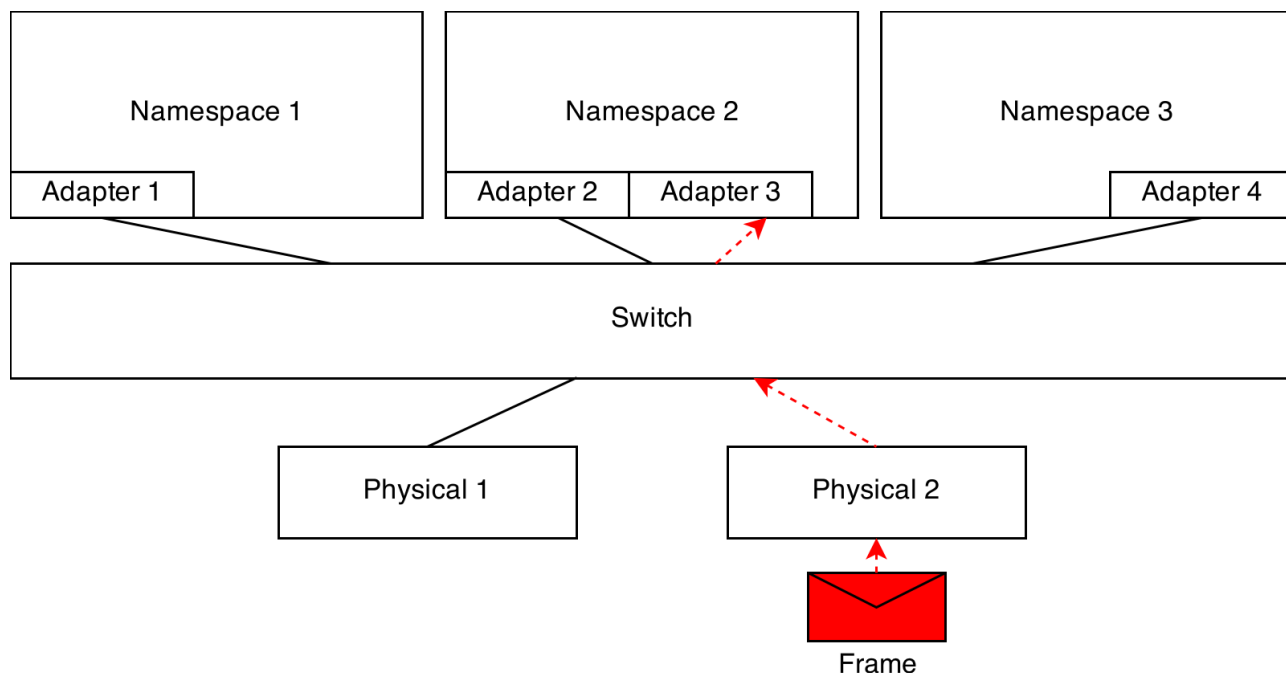
FIGURE 19 – Diagramme de lien avec une Virtual Ethernet (instance d'une LAN_ADAPTER + IP_INTERFACE)



5.2.4 Implémentation d'un switch pour connecter les namespaces

Pour pouvoir rediriger une trame dans son namespace adéquat, il va falloir switcher les trames logiquement. En effet, chaque namespace ayant des interfaces de niveau 2 (avec des adresses MAC uniques), l'aiguillage va se faire à ce niveau là.

FIGURE 20 – Switching de trames



Cette implémentation est l'équivalent sous Linux de la commande :

```
« ip link add veth0 type veth peer name veth1 »
```

Sous linux, cela crée une paire d'interface (une pipe, donc tout ce qui entre d'un côté ressort de l'autre). Pour connecter un namespace avec le monde extérieur, on place un bout du pipe dans le namespace, et l'autre bout du pipe dans la partie réseau globale (physique).

Dans notre cas, on va connecter une interface virtuelle (qui elle est attaché à un namespace) à un switch virtuel qui va s'occuper de switcher les trames vers différents endroits (un autre namespace ou le monde extérieur). Le switch n'est réellement qu'un seul gros switch mais il connaît les différents namespaces (« Virtual LAN ») existants et peut donc faire la part des choses. Voyons maintenant l'implémentation dans le code.

La première implémentation qui a été faite est d'intégrer un switch virtuel dans la plus basse couche de la réception de trames dans `tcpip.sys` (ReceiveHandler). Le travail du switch était simplement de comparer l'adresse MAC de destination de la trame et d'adapter l'interface logique source dans le code pour que les traitements soient cohérents pour la suite. L'idée de base était là, mais l'implémenter dans `tcpip.sys` n'est pas la solution. A terme, l'idée est d'intégrer un « vrai » (plus complet) switch virtuel (comme par exemple Open vSwitch) qui lui, dispose de beaucoup plus d'options (802.1Q, etc.). Le but n'étant pas de réécrire un vrai switch, on va uniquement supporter le switching de base, il y aura donc un certain manque de support plus loin, nous allons voir ça.

Listing 12 – Prototype de la fonction NDIS s'occupant de la réception d'un paquet

```
INT NTAPI ProtocolReceivePacket(  
    NDIS_HANDLE BindingContext, // LAN_ADAPTER  
    PNDIS_PACKET NdisPacket    // NDIS_PACKET  
);
```

Pour déplacer l'implémentation hors de `tcpip.sys` (pour pouvoir changer le switch facilement sans devoir modifier le coeur du code), l'idée que j'ai retenue est de faire un deuxième ProtocolDriver qui s'occuperait de switcher puis de renvoyer le résultat à `tcpip.sys` comme si de rien n'était pour lui. C'est ce procédé qui est utilisé sous Windows par les switches virtuels les plus utilisés, comme le « VirtualBox Bridged Networking Driver » qui permet d'utiliser des interfaces virtuelles dans VirtualBox.

Etant donné que tous les ProtocolDrivers bindés à une interface reçoivent la même trame¹⁰, sous Windows, il suffit de décocher le « Protocol TCP/IP » de l'interface physique et de laisser le « Protocol Switch » pour arriver à un switching de plus bas niveau. Cependant, cette partie n'est pas implémentée dans ReactOS. C'est un vrai casse tête pour pouvoir gérer facilement le binding d'une interface réseau (l'interface graphique ne supporte même pas la possibilité de cocher ou décocher un ProtocolDriver)¹¹.

10. C'est comme ça que fonctionne le driver WinPcap qui permet de sniffer le réseau sous Windows (utilisé par Wireshark, par exemple). En s'installant comme ProtocolDriver, le driver reçoit une copie de tout le trafic indépendamment du fonctionnement de TCP/IP. Cela permet également au driver de pouvoir injecter des paquets aux interfaces auxquelles le ProtocolDriver est bindé

11. Voir screenshots section 11.6.2 page 83

FIGURE 21 – Diagramme montrant la nécessité du switch

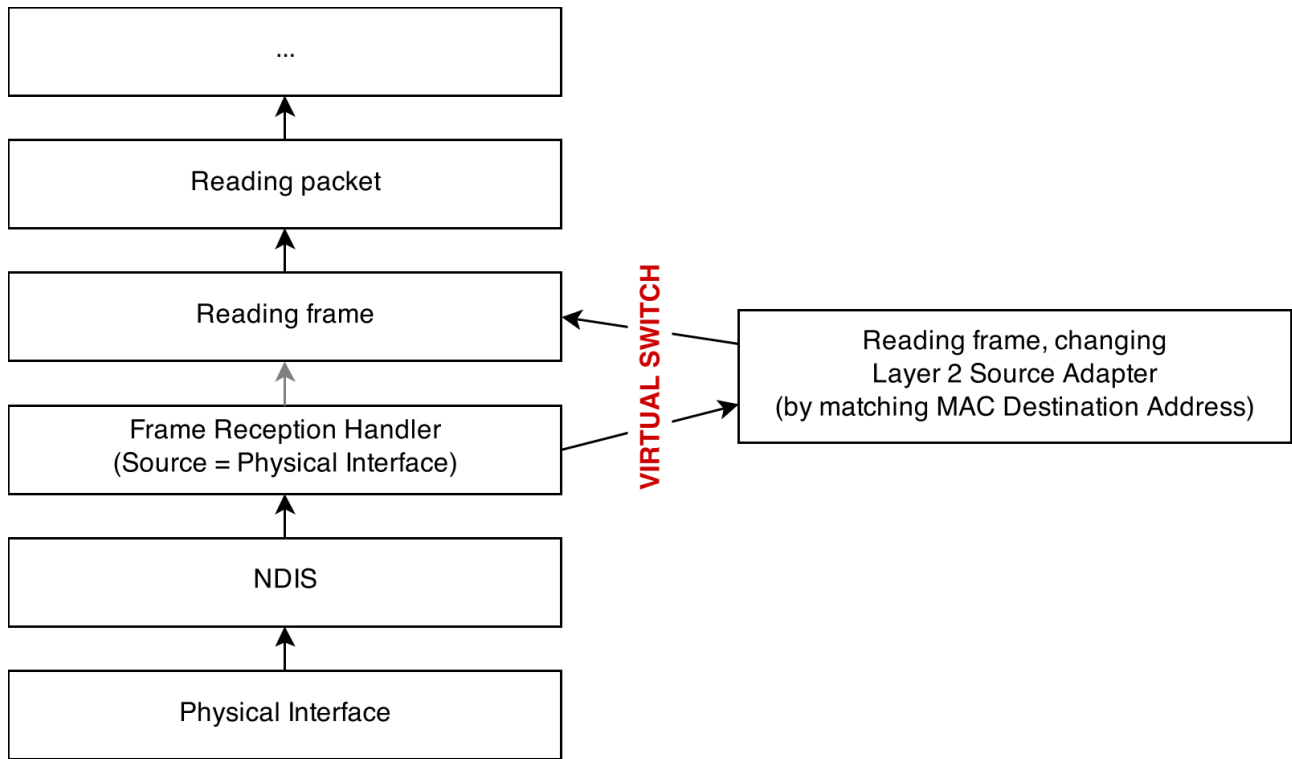
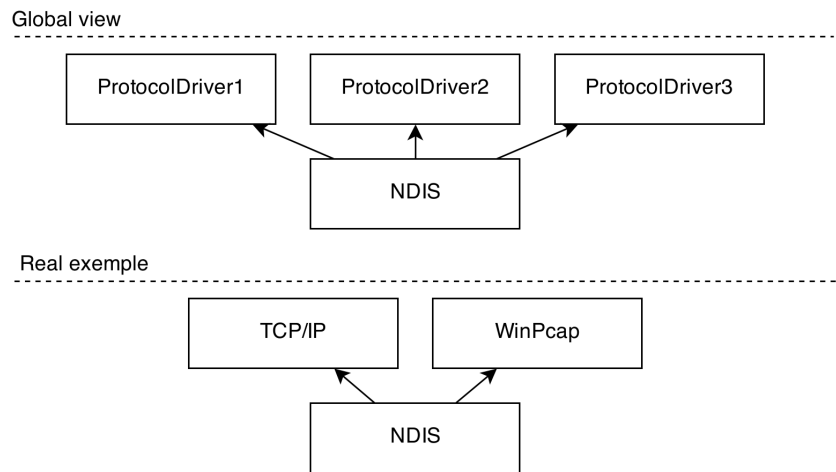


FIGURE 22 – Fonctionnement d'un ProtocolDriver NDIS

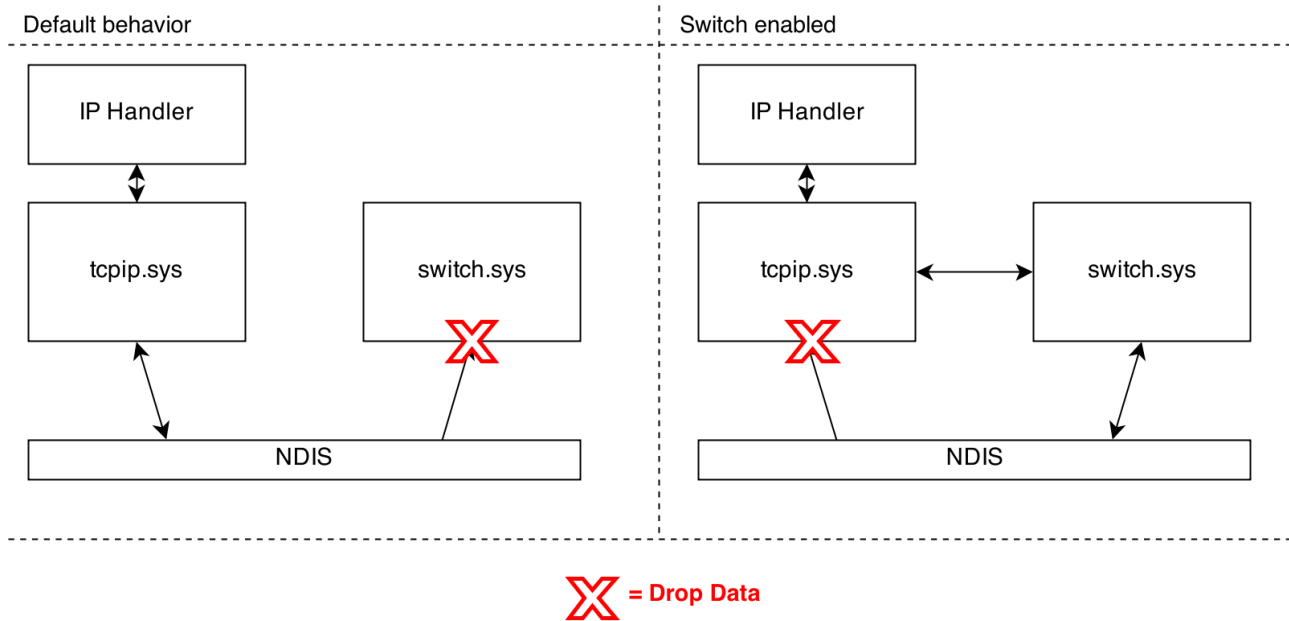


Pour pallier à ce problème, je suis parti sur une implémentation de switch qu'on peut activer et désactiver à la volée (via [unshare enable switch](#) et [unshare disable switch](#)).

Pour connecter le switch avec la pile TCP/IP existante, il faut faire communiquer les deux drivers entre eux. L'idée la plus simple et propre est de transférer le paquet entre l'un et l'autre via un [IoCallDriver](#), cependant ça veut dire qu'à **CHAQUE** trame reçue, un appel à l'Object

Manager du kernel va être fait, créer une interruption logiciel, ajouter une entrée dans une queue d'exécution, etc. Bref, quelque chose de très lourd pour simplement forwarder un paquet entre deux drivers.

FIGURE 23 – Diagramme d'utilisation de switch.sys



Une autre idée pour implémenter une communication entre les deux est de compiler et linker le switch avec la bibliothèque IP du système. De cette façon, non seulement je peux utiliser les mêmes structures de données, mais en plus je pourrais partager des variables globales entre `tcpip.sys` et `switch.sys` par le biais de la bibliothèque... mais ce n'est malheureusement pas le cas. En effet, la bibliothèque est liée statiquement aux drivers, donc chaque driver à sa propre instance. Une autre solution est donc à trouver.

L'un des avantages de se trouver en kernel space c'est qu'on est beaucoup moins restreint quant à l'utilisation de la mémoire (il n'y a pas cette protection du heap comme en user-mode qui empêche au code d'accéder à du contenu qui ne lui appartient pas). Du coup, avec une méthode peu commode, j'ai créé une structure qui va stocker des pointeurs de `tcpip.sys` et les envoyer à `switch.sys` via une communication drivers classique.

Listing 13 – struct SWITCH_NOTIFY

```
typedef struct _SWITCH_NOTIFY {
    PCHAR TCPIPEnabled;
    VOID (*LANTransmit)(PVOID, PNDIS_PACKET, UINT, PVOID, USHORT, [...]);
    NDIS_STATUS NTAPI (*ProtocolReceive)(NDIS_HANDLE, NDIS_HANDLE, [...]);
    INT NTAPI (*ProtocolReceivePacket)(NDIS_HANDLE, PNDIS_PACKET);
    PLIST_ENTRY AdapterListHead;
    PKSPIN_LOCK AdapterListLock;
    NDIS_HANDLE GlobalPacketPool;
    NDIS_HANDLE GlobalBufferPool;
} SWITCH_NOTIFY, *PSWITCH_NOTIFY;
```

On voit directement à la longueur du code, qu'il s'agit de pointeurs de fonctions. Le reste, ce sont des pointeurs classiques vers des variables ou même une copie de certaines variables (comme les `NDIS_HANDLE` qui ne sont pas nécessaires à déréférencer car fixes). Pour ce qui est de l'utilisation :

- **TCPIPEnabled** : un pointeur vers un flag qui active (ou pas) `tcpip.sys`. Ce flag est utilisé conjointement avec `SwitchEnabled` qui se trouve dans le code de `switch.c`
- **ProtocolReceive** : pointeur vers la fonction `ProtocolReceive` de `tcpip.sys`
- **ProtocolReceivePacket** : pointeur vers la fonction `ProtocolReceivePacket` de `tcpip.sys`
- **AdapterListHead** : pointeur vers la liste chaînée des `LAN_ADAPTER` de `tcpip.sys`
- **AdapterListLock** : pointeur vers le verrou (mutex) de la liste chaînée

- **GlobalPacketPool** : copie du handle `PacketPool` de `tcpip.sys` utilisé par NDIS (pour émettre un paquet)
- **GlobalBufferPool** : copie du handle `BufferPool` de `tcpip.sys` utilisé par NDIS (également pour émettre un paquet)

Le driver `tcpip.sys` est adapté pour que dès qu'une modification d'un de ces champs est provoquée, le driver recrée cette structure et la ré-envoie au switch pour le tenir informé des nouvelles modifications.

Une fois que le switch reçoit cette structure, il s'occupe de copier le contenu dans une variable globale. En plus de ça, il est possible que la mise à jour ait été déclenchée par l'ajout d'une nouvelle interface. Dans ce cas, par sécurité, si le switch est activé (si le flag `SwitchEnabled` est à `TRUE`), on va itérer sur toutes les interfaces et "rerouter" les fonctions de réception et d'émission de l'interface vers le switch.

Listing 14 – Reroutage des fonctions de réception et de transmission de paquets de `tcpip.sys` vers le switch virtuel

```

if(SwitchEnabled == TRUE) {
    DbgPrint("Switch: rerouting interfaces to switch\n");
    CurrentEntry = Linker.AdapterListHead->Flink;

    while (CurrentEntry != Linker.AdapterListHead) {
        Current = CONTAINING_RECORD(CurrentEntry, LAN_ADAPTER, ListEntry);
        ((PIP_INTERFACE) Current->Context)->Fallback =
            ((PIP_INTERFACE) Current->Context)->Transmit;
        ((PIP_INTERFACE) Current->Context)->Transmit = SwitchTransmit;
        CurrentEntry = CurrentEntry->Flink;
    }
}

```

A partir de ce moment là, tout le trafic sortant va passer d'abord par le switch avant de passer par TCP/IP et le trafic entrant passera par le switch avant de passer dans TCP/IP. Il ne reste plus qu'à faire le traitement des paquets.

A noter : même si il n'y a qu'une seule instance du switch virtuel, d'un point de vue logique, nous disposons d'un switch par « Virtual LAN ». En effet, dès que deux interfaces (virtuelles ou pas) sont connectées/attachées l'une à l'autre, un point de switching entre les deux s'établit (le

switch fait le rapprochement en comparant le handle NDIS) ¹².

Lors de la demande de transmission d'un paquet, on reçoit en paramètres le paquet IP et un pointeur vers une adresse (MAC) de destination.

5.3 Lien entre la couche utilisateur et le driver TCP/IP

Maintenant que les couches 2 et 3 sont isolées dans le kernel, il faut voir comment faire le lien avec la partie utilisateur. Je suis parti d'ipconfig (la commande donne l'adresse IP et MAC des interfaces installées sur le système, donc je suppose que les appels que ce programme fait, sont directement liés au code que j'ai modifié plus haut).

5.3.1 Utilitaires de gestion : ipconfig.exe, route.exe, arp.exe, ping.exe, ...

Le code source de « ipconfig » se trouve dans `/base/applications/network/ipconfig`. Le code semble avoir été réécrit from scratch mais utilise bien « iphlapi » (comme sous Windows), pour communiquer avec la couche réseau du système.

Dans le header du code, un commentaire dit directement que « renew » et « release » ne sont pas implémentés. Nous sommes au sommet de la couche d'appel, et on se rend compte d'une chose : il manque des fonctionnalités, et même de base, mais bon, le principal semble fonctionner.

Globalement, l'affichage des interfaces ainsi que leurs état, adresse, etc. se font via l'appel de la fonction `GetAdaptersInfo` qui se trouve dans la `iphlpapi`. D'après MSDN, cette appel retourne une liste chaînée de structures `IP_ADAPTER_INFO`.

Le contenu de la structure de `IP_ADAPTER_INFO` comparé à ce qu'on vient de voir (la séparation layer 3 et 2) n'est plus respecté, c'est une structure prévue pour répondre à l'utilisateur le plus de choses possible, du coup on se retrouve avec un mix de tout :

12. La bonne méthode serait de d'abord créer un Virtual LAN, puis d'y attacher des interfaces et non pas de créer un Virtual LAN implicitement lors d'une connexion

Listing 15 – struct IP_ADAPTER_INFO (repris de la MSDN)

```
typedef struct _IP_ADAPTER_INFO {
    struct _IP_ADAPTER_INFO *Next;
    char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD Index;
    UINT Type;
    UINT DhcpEnabled;
    PIP_ADDR_STRING CurrentIpAddress;
    IP_ADDR_STRING IpAddressList;
    IP_ADDR_STRING GatewayList;
    IP_ADDR_STRING DhcpServer;
    BOOL HaveWins;
    IP_ADDR_STRING PrimaryWinsServer;
    time_t LeaseExpires;
    // [...]
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

On voit bien qu'on dispose d'un tas de choses totalement hétérogène d'un point de vue isolation : MAC, IP, DHCP, DNS, Gateway, etc. On a donc une concentration et une centralisation des appels qui se font via [iphlpapi](#).

5.3.2 IP Helper API (iphlpapi)

Cette API permet de questionner le système à propos de sa configuration réseau. Elle ne permet pas d'envoyer ou recevoir des paquets IP, mais bien uniquement de récupérer (ou configurer) des informations.

Par exemple, la fonction [getInterfaceInfoSet](#) permet d'établir une liste des interfaces réseaux disponibles. En réalité, il s'agit d'un helper pour l'utilisation de TDI. Pour établir la liste des interfaces, iphlpapi va demander à TDI de retourner une liste d'entités puis cette liste va être parcourue et analysée. C'est également dans cette partie du code que l'interface de loopback va être ignorée, c'est pourquoi elle n'apparaît pas dans ipconfig (alors qu'elle existe bel et bien).

Quelle que soit la partie du code de iphlpapi, on remarque un appel récurrent qui est [tdiGetEntityIDSet](#).

5.3.3 TDI (Transport Dispatch Interface)

TDI est un protocole utilisé pour communiquer avec la couche de transport de la stack réseau de Windows. Les « Transport Providers » sont l'implémentation de protocoles réseau tel que TCP/IP, NetBIOS, et AppleTalk. Durant la compilation et le linkage d'une application user-mode, un client TDI est intégré dans le code. Ce client permet de faire une passerelle pour les API user-mode des ProtocolDrivers. Typiquement, les commandes TDI sont : TDI_SEND, TDI_CONNECT, TDI_RECEIVE.

On trouve TDI à trois endroits dans l'arborescence :

- `/drivers/network/tdi/tdi/`
- `/drivers/network/tcpip/` (avec le reste de TCP/IP)
- `/lib/tdilib/`

La partie qui se trouve dans `drivers/tdi` ne contient que du code non-implémenté, certainement pour garder une compatibilité de symbols avec des applications qui l'utilisent (la partie NetBios n'est absolument pas gérée par exemple).

La `tdilib` par contre est très petite mais implémentée. Il n'y a pas grand chose derrière. Lors d'une interrogation via TDI, les étapes suivantes sont effectuées :

- Ouverture de l'objet `\device\Xxxx`¹³
- Appel de `IOCTL_TCP_QUERY_INFORMATION_EX` sur le driver avec les paramètres qui ont été fournis à l'appel de TDI (le type d'entité demandé, etc.)
- Retour de la réponse du driver directement à l'appelant (sans même lire la réponse ou la formater)

L'objet `\device\tcp` est géré dans `tcpip` et la liste des entités (TDI Entities) aussi. Étant donné que toutes les informations sont transmises via ces entités, pour isoler la réponse côté user-mode, il suffit de modifier la liste d'entités retournée.

Malheureusement, encore une fois, dans le code de `tcpip.sys`, la liste des `TDIEntityID` se retrouve dans une variable globale au code, et cette variable est utilisée pas mal de fois dans le code. De plus, la structure `TDIEntityID` n'est pas un type défini par ReactOS, mais bien par l'implémentation de TDI, plus d'informations à propos de cette structure se retrouvent dans la documentation

13. Variable en fonction de la destination : Tcp, Udp, Raw, ...

MSDN¹⁴. Par sécurité (et propreté, bien que la propreté puisse être subjective vu ce qui arrive...) j'ai décidé de ne pas modifier cette structure, étant donné qu'elle est explicitement décrite (ce n'est pas le cas de la structure `EPROCESS` que j'ai modifiée, car elle n'est que partiellement décrite dans la documentation, certaines parties sont cachées).

Pour pouvoir disposer de plusieurs listes d'entités, j'ai tout simplement créé une liste qui contient des listes d'entités. Il y a deux variables globales utilisées : `EntityList` et `EntityCount`. La première variable est un vecteur d'entités, la deuxième est un entier qui contient le nombre d'éléments du vecteur. Pour ne pas modifier le code existant, j'ai redéfini ces deux variables par un appel de fonction qui va retourner la liste correspondante au namespace en cours.

Listing 16 – Adaptation de la liste d'entités TDI

```
/* Override original global names with functions */
#define EntityList GetEntityList(IoGetCurrentProcessNs())
#define EntityCount GetEntityCount(IoGetCurrentProcessNs())

TDIEntityInfo *GetEntityList(ULONG NamespaceId);
ULONG GetEntityCount(ULONG NamespaceId);
ULONG SetEntityCount(ULONG Count);

NTSTATUS TdiNewNamespace(ULONG NamespaceId);
NTSTATUS TdiRemoveNamespace(ULONG NamespaceId);
```

Cette méthode (bien que très agressive) fonctionne très bien et propose plusieurs avantages :

- Les appels déjà existants à `EntityList` sont toujours fonctionnels
- On peut débayer l'appel de la liste vu qu'à chaque accès, une fonction est appelée
- La liste peut être adaptée plus tard, en changeant juste la fonction appelée ou en modifiant la fonction `GetXxxxxx` directement.

Pour stocker ces différentes listes, on va créer deux nouvelles structures qui vont contenir les informations nécessaires pour garder différentes listes de `TDIEntityInfo`.

14. [http://msdn.microsoft.com/en-us/library/bb432492\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb432492(v=vs.85).aspx)

Listing 17 – Structure de données pour stocker plusieurs listes d’entités TDI

```
typedef struct _TDINamespaceInfo {
    LIST_ENTRY ListEntry;      /* Linked list Entry */
    ULONG NamespaceId;        /* Namespace ID */
    TDIEntityInfo *tdiEntry;  /* Real EntityList */
} TDINamespaceInfo;

typedef struct _TDINamespaceCount {
    LIST_ENTRY ListEntry;      /* Linked list Entry */
    ULONG NamespaceId;        /* Namespace ID */
    ULONG CountValue;         /* Real EntityCount */
} TDINamespaceCount;
```

Grâce à ces deux structures, on peut identifier chaque [EntityList](#) et [EntityCount](#) par un ID qui sera le numéro du namespace.

Pour ce qui est des deux fonctions (qui remplacent les variables) [GetEntityList](#), il ne s’agit plus que d’une itération sur la liste des [EntityList](#) et de retourner celui qui correspond au namespace demandé.

Listing 18 – Fonctionnement de [GetEntityList](#)

```
TDIEntityInfo *GetEntityList(ULONG NamespaceId)
{
    TDINamespaceInfo *EntryInfo;
    PLIST_ENTRY Entry = __EntityListNS.Flink;

    while (Entry != &__EntityListNS) {
        EntryInfo = CONTAINING_RECORD(Entry, TDINamespaceInfo, ListEntry);

        if (EntryInfo->NamespaceId == NamespaceId)
            return EntryInfo->tdiEntry;

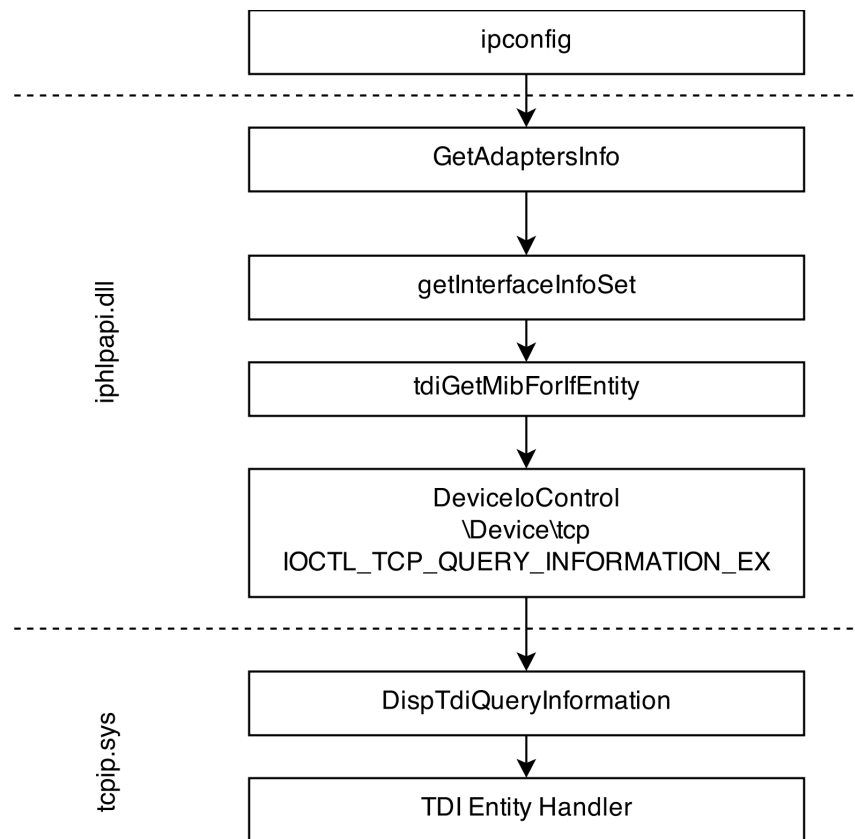
        Entry = Entry->Flink;
    }

    return NULL;
}
```

Pour ce qui est de `TdiNewNamespace` et `TdiRemoveNamespace`, rappelez-vous, ces fonctions sont appelées lors de la création d'un namespace (depuis `process.c`).

Nous avons le point clé de l'isolation : les applications de configuration et de management en user-mode (`ipconfig`, `arp`, `route`, ...) utilisent tous `iphlpapi` (qui utilise TDI). En réalité, tout le user-mode utilise TDI pour communiquer avec la couche réseau du système (également Winsock), c'est grâce à cette modification dans TDI qu'on peut isoler une stack TCP/IP sans aller plus loin que la couche 3. Grâce à ça, l'implémentation actuelle de UDP, TCP, etc. n'a pas à être modifié.

FIGURE 24 – Pile d'appels depuis `ipconfig` jusqu'à `tcpip.sys`

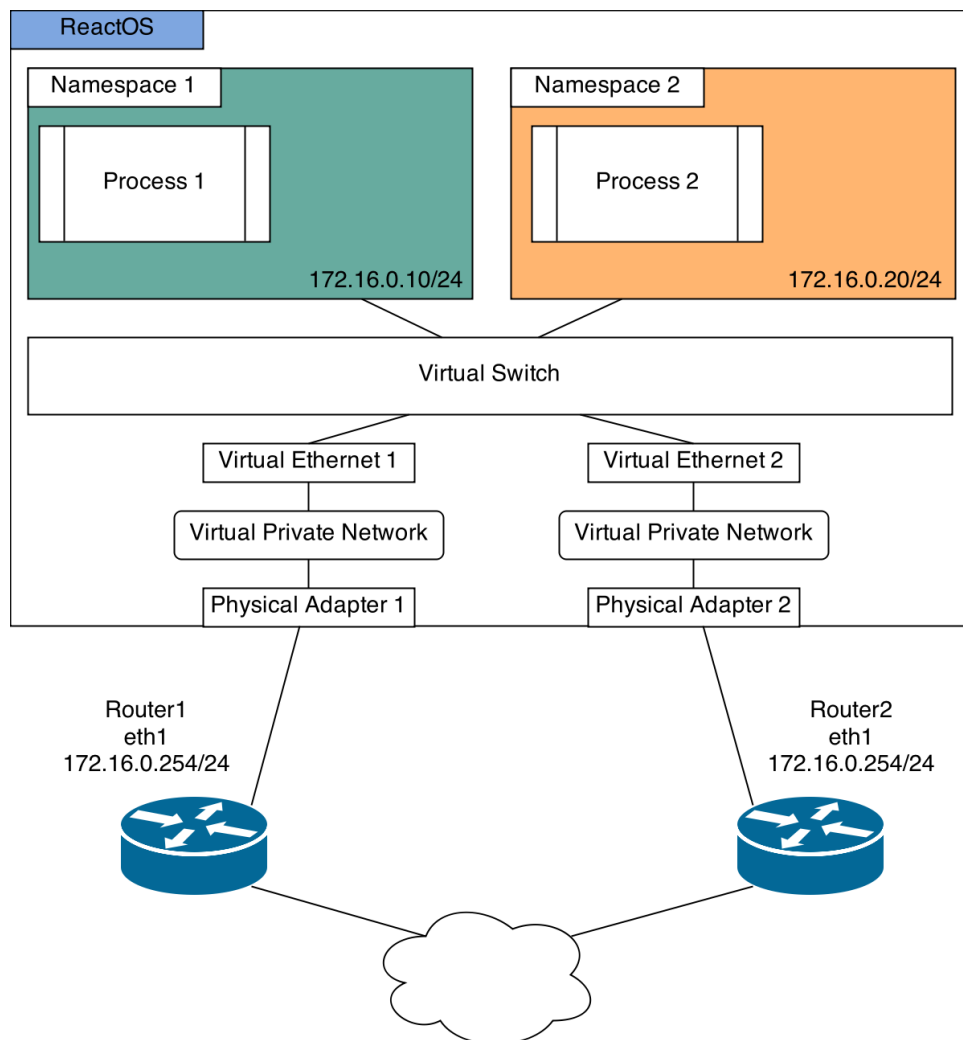


6 Topologies de test

6.1 Overlapping d'adresses IP de destination

Une première démonstration du fonctionnement de l'isolation est un exemple avec de l'overlapping d'IP de destination. En effet dans cet exemple on va avoir deux processus (ex : deux `cmd.exe`) dans deux namespaces différents qui seront tous les deux attachés à deux interfaces physiques différentes. Chacune des interfaces physiques est connectée à un routeur derrière, qui ont tout les deux l'adresse IP `172.16.0.254/24`. Dans un environnement classique, il ne peut y avoir qu'une seule route vers `172.16.0.254`, cet exemple montre bien qu'on dispose de deux tables de routage et deux tables ARP différentes pour les deux processus.

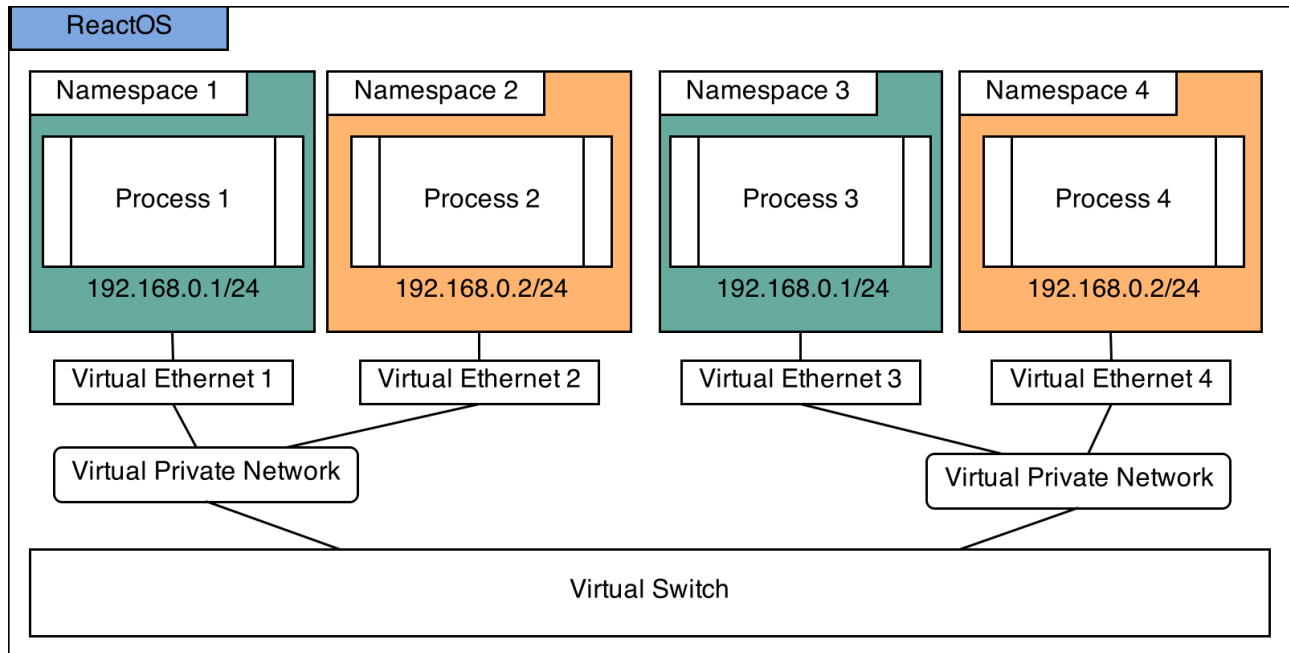
FIGURE 25 – Topologie de test avec overlapping d'adresses IP de destination



6.2 Overlapping d'adresses IP en communication inter-namespaces

Une autre démonstration est l'isolement inter-namespaces. Dans cet exemple on va faire 4 namespaces qu'on va regrouper en 2 isolations logiques. On va inter-connecter 2 namespaces entre eux, et les deux autres entre eux. Pour complexifier le tout, on va faire de l'overlapping dans les deux namespaces logiques. Le diagramme est plus que nécessaire pour comprendre le principe :

FIGURE 26 – Topologie de test avec overlapping d'IP inter-namespaces



7 Conclusion

7.1 Technique

En partant d'un OS écrit « from scratch », en analysant le fonctionnement et en lisant le code source du système, je suis parvenu à implémenter un Proof-Of-Concept relativement poussé d'une isolation complète d'une stack TCP/IP : les interfaces réseaux, la table de routage et la table ARP sont isolées.

Du côté du user-mode, les réponses des applications `ipconfig.exe`, `route.exe`, `arp.exe` sont bien isolées également. Un « `ipconfig` » ne donnera que la liste des interfaces réseaux de son namespace (et leurs adresses IP et MAC). Un « `route print` » n'affiche que les routes de son namespace. Pareil pour la commande « `arp -a` » qui donnera la table ARP de son namespace, sans interférer avec les autres.

En plus de ça, l'isolation va à un tel point que je peux faire une topologie qui supporte de l'overlapping d'adresses aussi bien sources que de destinations. En effet, dans deux namespaces différents, je peux ping la même ip (destination) et voir que le traitement du paquet ne se fait pas de la même façon (routage précis sur base de l'application source).

J'ai écrit un simple (mais suffisant pour l'instant) switch virtuel qui s'occupe d'aiguiller les trames que le système reçoit, de façon à avoir différents domaines de broadcast (sur une certaine limitation : il n'est pas possible d'avoir plusieurs domaines de broadcast sur une interface physique, cela demande un support de tagging de paquet, comme fait le 802.1Q).

Pour finir, j'ai établi un canal de communication entre les drivers (nouveaux et existants) et le user-mode. J'ai pu traverser toute la couche d'un OS (user-mode vers le kernel-mode, en passant par un driver) et toute la stack IP d'un système (en se limitant aux couches de liaison et transport).

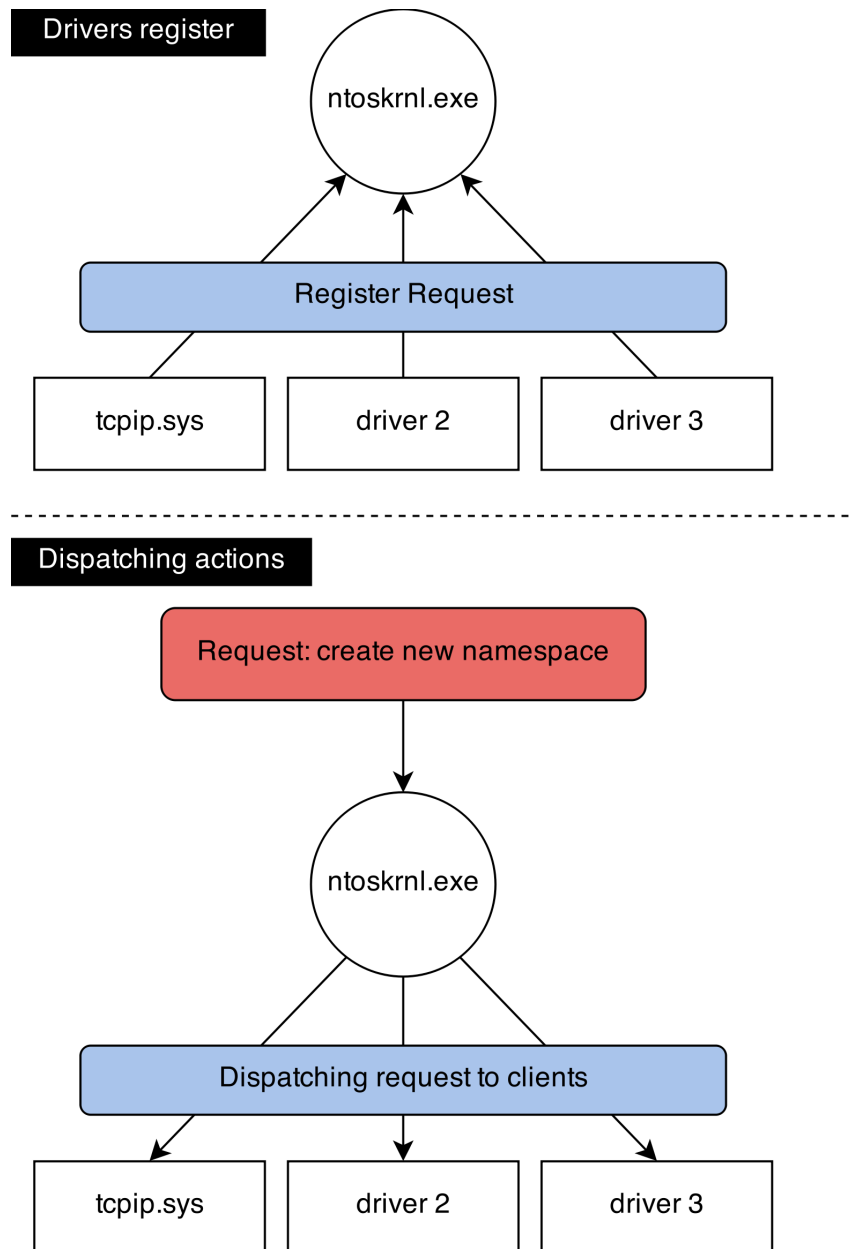
J'ai pu intégrer un principe fonctionnel et utilisable dans une topologie de test, il reste cependant pas mal de travail à faire pour rendre ce système parfaitement fonctionnel et utilisable à grande échelle (à commencer par remplacer mon pauvre switch de base par un switch plus évolué).

7.2 Recommandations pour la suite

Un grand nombre de points sont à améliorer ou corriger, certains sont dûs à une limitation d'implémentation, d'autres à des proof-of-concept qui sont restés dans le code et n'ont jamais été améliorés.

- Switch virtuel : le `switch.sys` qui a été écrit spécialement pour les namespaces n'est pas à garder. En effet il ne permet qu'un switching très sommaire (sur base d'adresses MAC uniquement) et est très loin d'être optimisé. De plus il a une forte dépendance à `tcpip.sys` et à la lib ip du système. L'idée qui a été suggérée par mon maitre de stage est d'utiliser un portage de openvswitch pour Windows. Cependant à l'heure actuelle, l'implémentation est partielle. Pour profiter pleinement des namespaces, un switch supportant des VLAN semble indispensable pour pouvoir isoler correctement la couche 2 et 3.
- Le système utilise globalement des listes linéaires pour stocker toutes les tables existantes. Dans beaucoup de cas liés aux namespaces, ces listes sont itérées à chaque réception de paquets. L'utilisation d'un meilleur algorithme ou tout simplement de hashtable pour stocker ces tables semble incontournable pour avoir de meilleures performances.
- La communication entre le kernel et le driver réseau se fait par l'objet `\Device\NamespaceNetwork`, qui est initialisé dans le driver `tcpip.sys`. Cette pratique est une relique d'implémentation. Mon but a été de faire communiquer le kernel avec `tcpip.sys` rapidement et c'est la première solution qui m'est venue à l'esprit, cependant maintenant que l'implémentation est plus aboutie, on se rend compte que cette méthode n'est pas optimale. Seul `tcpip.sys` peut bénéficier d'une mise à jour du kernel (être appelé depuis le kernel). La solution serait de créer l'objet `\Device\NamespaceNetwork` dans `ntoskrnl.exe` (dans la partie `networknamespace.c`) et d'y ajouter une liste de Listeners. Chaque partie du système (driver ou autre) ayant un rapport avec le réseau pourrait alors s'enregistrer comme « client network namespace » au-près du kernel. De ce fait, lors d'une demande de création d'un namespace (par exemple), les différents drivers qui ont une part à jouer dans le namespace (`tcpip.sys` doit allouer de nouvelles listes dans son code par exemple) seraient notifiés depuis le kernel.

FIGURE 27 – Enregistrement des drivers dans le kernel et dispatching d’actions



- Remplacer le hard-code de détection du PID `0x04` par une fonction qui vérifierait de façon plus générique si le processus en cours d’exécution est le processus `SYSTEM` ou pas. Voir section 5.1.7 page 41.
- Faire un tirage aléatoire de l’adresse MAC de la Virtual Ethernet autrement qu’en tirage totalement aléatoire sur les 48 bits. Par exemple, VirtualBox a son propre OUI et donc toutes leurs adresses MAC ont le même préfixe. Il est possible qu’un préfixe existe pour ce genre d’utilisation mais je n’en ai pas trouvé (certaines implémentations d’interfaces TAP font aussi un tirage aléatoire sur les 48 bits, c’est pourquoi j’ai laissé ça ainsi).

- Actuellement, toute la configuration et le fonctionnement des namespaces est volatile. Windows utilise sa base de registre pour y sauver toutes sortes de configurations, etc. Au lieu de faire un script batch au démarrage de la machine, une implémentation de la sauvegarde de la configuration (et son chargement au boot) dans la base de registre pourrait être un point intéressant comparé au fonctionnement général de Windows.
- Implémenter complètement le support de l'isolation des processus (supporter le fait d'avoir plusieurs PID les mêmes simultanément, ...)
- Le portage de la commande « ip » sous Linux, du package « iproute2 ». Ce package et la commande « ip » supporte presque tout l'aspect de gestion réseau sous Linux. (ip address, ip link, ip route, ip rule, ip neighbor, ip tunnel, etc.)
- Faire une extension pour « netsh » qui permettrait de gérer les namespaces depuis la ligne de commande. En effet, « netsh » est prévu pour supporter des extensions par l'ajout de fichiers `.dll`, un module qui s'occuperait de l'attachement/détachement des interfaces à un namespace depuis « netsh » semble un choix intéressant.
- Porter le code et le fonctionnement de ReactOS vers Windows pour en faire une solution comme « Parallels Containers for Windows », mais avec du code Open Source. Il ne sera plus question de modifier le fonctionnement du kernel par contre, mais bien d'adapter le fonctionnement en faisant une sur-couche au système existant.
- Développer un plugin pour GNS3. Ce programme offre une GUI qui permet de faire une topologie très simplement. A la base, la GUI sert à configurer des routeurs Cisco pour configurer une topologie avec l'émulateur [dynamips](#), mais leur interface graphique est adaptable. On peut imaginer utiliser l'interface de GNS3 pour configurer les namespaces de la machine et ainsi construire la topologie et la connexion des containers graphiquement.
- Hyper-V utilise un switch virtuel qui a été développé par Microsoft. Ce switch est utilisé partout dans l'hyperviseur de Microsoft et supporte déjà pas mal de fonctionnalités (VLAN, etc.). Il serait intéressant de remplacer mon implémentation du switch virtuel par la leur et donc offrir une couche de virtualisation (containers) par dessus leur switch.

8 Conclusion personnelle

8.1 A propos de mes études

Étant en option réseau à la Haute École de la Province de Liège, ce sujet est directement lié à mes études. Bien que la majorité des cours soient basés sur des langages de hauts niveaux, les cours de C et **surtout** de réseau m'ont permis d'arriver plus ou moins facilement à bout de ce travail.

La programmation kernel n'est pas vraiment abordée dans mon cursus, j'ai donc dû trouver et apprendre par moi-même une grande partie du fonctionnement de Windows. J'ai toujours été intéressé et passionné par Linux (le kernel et le système GNU/Linux en général), j'ai donc pu me baser sur ce que je connaissais pour le transposer dans une logique Windows.

Concrètement, mon développement dans le kernel s'achève avec **84** fichiers modifiés, **4130** lignes ajoutées et **550** lignes supprimées (il s'agit d'un [git diff](#) entre mon checkout du SVN ReactOS et mon dernier commit dans mon git local)

8.2 Sujet du TFE

Ce sujet est particulièrement intéressant car il n'existe aucune alternative libre ou même native à ce que j'ai conçu/écrit. Le fait de faire quelque chose de neuf et dans un domaine très recherché à l'heure actuelle (la virtualisation) m'a vraiment plu et m'a appris beaucoup de choses, principalement à propos du kernel Windows NT.

Ca me rassure réellement de voir qu'il reste encore des domaines et des sujets où les langages de hauts niveaux n'ont pas vraiment leurs places. Il n'est pas vraiment concevable à l'heure actuelle de faire un hyperviseur (faible en demande de ressources) en Java par exemple.

Je n'avais jamais eu l'occasion de tester réellement les namespaces sous Linux avant de commencer ce stage et je suis très content d'avoir pu découvrir cette partie de la virtualisation qui est pour moi un concurrent sérieux aux hyperviseurs de type 1 et 2 à l'heure actuelle où le partage de ressources et la rentabilité du matériel deviennent un point critique.

À l'heure actuelle où le « cloud » devient un mot à la mode et que tout le monde s'empresse

d'utiliser, cette technologie demande justement énormément de virtualisation (mettre en ligne du contenu dans un environnement sécurisé (et donc isolé)). Si il fallait faire une machine virtuelle par utilisateur d'un système, l'overhead global serait énorme et la rentabilité des ressource ne serait pas optimale.

8.3 Retour sur le stage

Je me suis vraiment bien amusé durant mon stage. L'ambiance générale de la boîte et dans l'openspace où je codais était vraiment sympa. De plus, je ne suis absolument pas quelqu'un « du matin », et Level IT est le genre de société qui offre une certaine flexibilité d'horaire qui me permet, je trouve, de bien travailler.

Le courant est (vite) très bien passé avec mon entourage. Je me suis intégré facilement à l'équipe et j'ai vraiment bien aimé développer dans des conditions qui me permettent d'utiliser mon propre laptop avec mon système d'exploitation et mon environnement de développement, sans être obligé et confiné à utiliser des logiciels précis.

9 Remerciement

Je souhaite tout d'abord remercier Pierre De Fooz pour m'avoir proposé ce stage. En effet, j'avais demandé si il était possible de trouver un stage qui impliquait d'écrire du C (n'étant vraiment pas intéressé par des langages de haut niveau comme C# et encore moins Java) dans le domaine du réseau. Il m'a trouvé ce sujet qui ne demande qu'à faire du C, dans un kernel et qui de plus est Open Source. C'est là exactement tout ce que je recherchais.

Je tiens à remercier également Olivier Hault, patron de Level IT, qui a toujours été derrière moi, m'a soutenu et donné des pistes à suivre pour arriver au but.

Le channel #reactos du serveur IRC irc.freenode.net, sur lequel j'ai pu avoir de l'aide des développeurs du projet ainsi que des contacts avec le coordinateur de projet actuel.

Denis Jasselette, Cyril Paulus et Christine Daniel pour la relecture de cet ouvrage.

Le channel #inpres du serveur IRC irc.maxux.net pour ses résidents, toujours prêts à aider et à troller : Raphael Javaux (RaphaelJ), Lionel Ancia (liBdot4), Benoît Dardenne (morte), Sacha Sokoloff (Darky), Laurent Doms (Pichet), Ghilan Onkelinxou (ghilan), Zoé (z03), Mathieu Lobet (omlet), Christophe De Carvalho (Zaibon), Kimberly Scott (woop), William Gathoye (wget), Anthony Binnici (Nemo), Thomas Van Gysegem (T4g1), Loïc Coenen (jt), Corentin Pazdera (nado), Sarah Sabry (Paglops).

Tous ceux que je n'aurais pas cité et qui m'ont permis d'arriver au bout de ce projet !

10 Informations complémentaires

- Ce document a été rédigé en LaTeX, compilé en PDF avec XeLaTeX sous GNU/Linux (Gentoo).
- Les diagrammes que j'ai réalisés ont été faits avec <https://www.draw.io>
- La coloration syntaxique de ce document est faite via *pygmentize* et *minted*. Pour bénéficier d'une coloration correcte des types, j'ai dû en ajouter un grand nombre au fichier source car Microsoft a eu l'excellente idée (je ne sais pas pourquoi à vrai dire) de redéfinir tous les types du C en majuscule et de ne respecter aucune convention vis-à-vis des customs types (par exemple faire des `typedef type_t`).
- Source des diagrammes de virtualisation : <https://fr.wikipedia.org/wiki/Virtualisation>

11 Annexes

11.1 Environnement de développement et compilation

La première chose à faire pour pouvoir utiliser ReactOS depuis les sources, est de cloner le SVN puis compiler le système complet. Un environnement de compilation (nommé RosBE) a été fait sur mesure pour compiler le système from scratch. Cet environnement comprend des scripts pour configurer des variables d'environnement et une toolchain complète (gcc, binutils, ...).

L'arborescence (principale) du code source de ReactOS se compose ainsi :

- `/base` : contient la source des applications userspace (ipconfig, calc, autochk, ...)
- `/boot` : contient le code du bootloader et les fichiers `.inf` permettant de construire un registre de base (pour le `livecd` ou le `bootcd`)
- `/cmake` : contient les fichiers de configuration pour la compilation du système complet
- `/dll` : contient le code des bibliothèques partagées de Windows (ntdll, directx, le control panel, ...) ainsi que des portages du monde libre (libpng, libjpeg, ...)
- `/drivers` : contient le code source des drivers qui ont été réécrits pour ReactOS (drivers IDE, USB, Serial, ...) dont notamment le driver TCP/IP et NDIS.
- `/include` : contient les `.h` du `ddk`, `nds`, `kernel` et tous les headers globaux du système
- `/lib` : contient la source de la RTL (RunTime Library) de Windows et des libs écrites spécialement pour ReactOS, dont la « libip »
- `/media` : contient tout ce qui ne se compile pas et qui est nécessaire au système (fichiers `inf`, `fonts`, `wallpapers`, ...)
- `/ntoskrnl` : contient le code de `ntoskrnl.exe` qui est réellement le coeur du système, c'est le kernel de Windows. Le contenu est structuré pour séparer la gestion de la mémoire, des processus, etc.

Le système se compile avec `cmake` et dispose d'une target `bootcd` ou `livecd`.

- `bootcd` : compile tout le système et crée un ISO bootable qui lance une installation du système (Windows XP like). Installer le système nécessite un disque dur, partitionner, un bootloader, etc. mais cette version permet d'utiliser le système en read-write
- `livecd` : compile tout le système et crée un ISO bootable qui lance le système sans l'installer. Cet ISO n'inclut pas l'installateur et n'est utilisable qu'en read-only. Cette version ne nécessite pas de disque dur. Le `livecd` est très pratique pour tester des modifications côté kernel et

drivers car durant le boot, le matériel est probé puis installé. Ca permet de partir d'un système fraîchement installé rapidement.

Tout mon travail s'est effectué sur la révision 62083 du SVN. Le code fait un peu plus de 3 millions de lignes de C et un peu moins de 2 millions de lignes de header. Ma machine de développement a été un laptop Acer 7730G, avec un Core 2 Duo à 2.2 Ghz et 4 Go de ram. La compilation du live cd met une grosse demi-heure à compiler.

11.2 Exécution et débogage

Pour l'exécution du système, j'ai utilisé la virtualisation et l'émulation. Mon laptop n'ayant pas de technologie de virtualisation vu l'âge du CPU, les solutions Xen, etc. n'étaient pas possibles (vu que c'est du Windows qui est virtualisé).

Mon choix s'est limité à VirtualBox et qemu :

- VirtualBox : hyperviseur de type 2, il m'a surtout permis d'utiliser le système avec plusieurs cartes réseaux virtuelles bindées à des Debian virtuels pour pouvoir analyser le trafic entrant et sortant.
- qemu : je n'ai pas réussi à virtualiser plusieurs interfaces réseaux bindées aux Debian mais l'utilisation de qemu permet un débogage du kernel avec l'utilitaire `gdb` à distance. Breakpoint, backtrace, chargements de symboles, etc. tout ce qu'il faut pour pouvoir avoir la main sur le kernel.

Dans le fichier de configuration `/cmake/config.cmake`, j'ai ajouté l'option `SEPARATE_DBG` qui permet de séparer les symboles de débogage des binaires, ce qui m'a permis de charger les symboles dans `gdb` et pouvoir faire du backtrace, des breakpoints, etc. dans le kernel et les drivers. Cette option ne se trouvait pas dans la config mais le flag `SEPARATE_DBG` existe et est implémenté dans le code qui s'occupe de compiler le projet.

11.3 lwIP (A Lightweight TCP/IP stack)

Ce projet¹⁵ est particulièrement intéressant car il s'agit d'une implémentation d'une stack TCP/IP simple mais complète, très légère et vraiment indépendante du système derrière, le code pouvant même tourner sur du matériel embarqué sans OS.

Le but de lwIP est d'avoir une implémentation la plus complète possible d'une stack TCP/IP tout en gardant une très faible consommation de ressources.

Les fonctionnalités principales de lwIP sont :

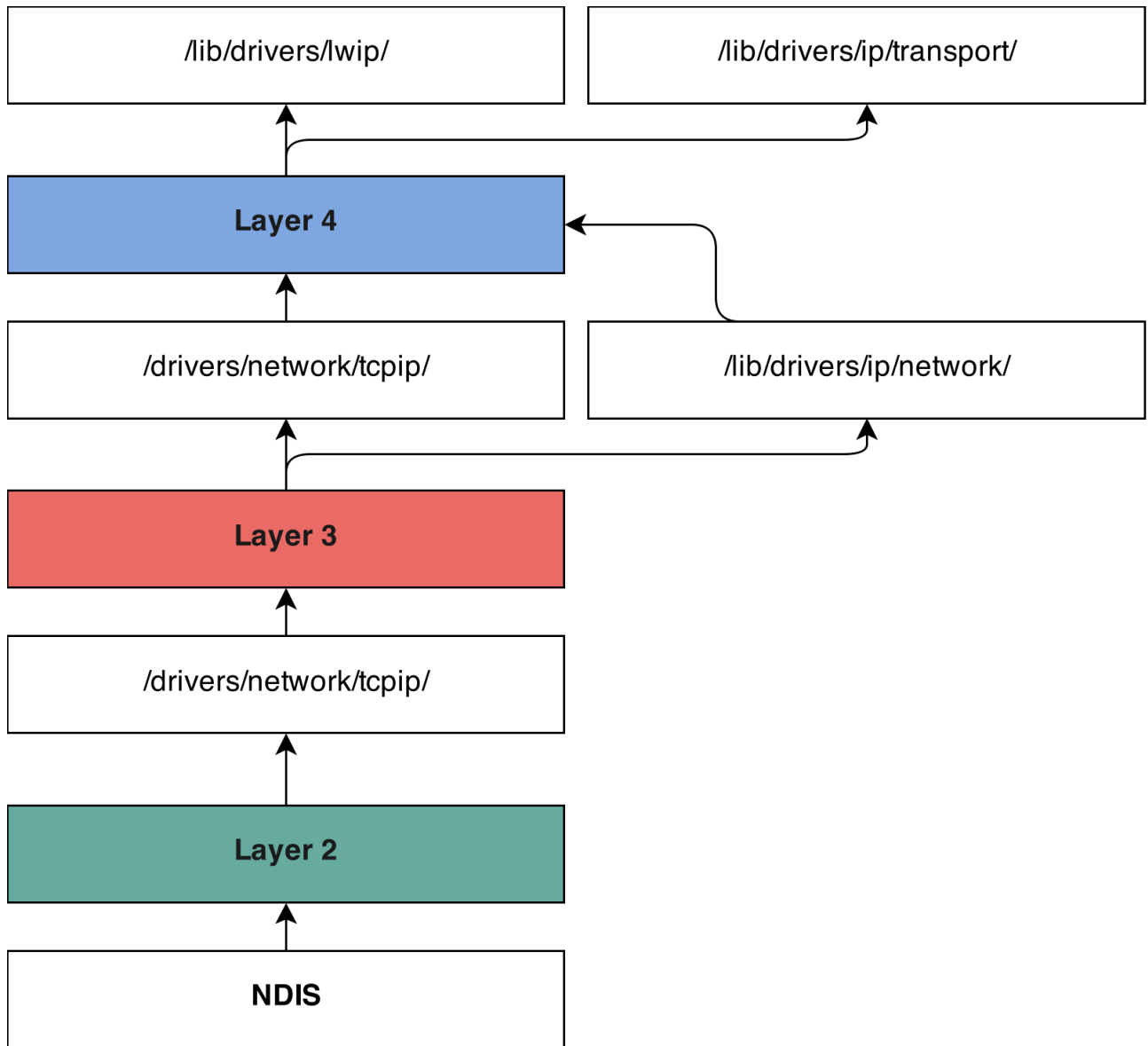
- Protocoles : IP, ICMP, UDP, TCP, IGMP, ARP, PPPoS, PPPoE
- Client DHCP, client DNS, AutoIP/APIPA (Zeroconf), agent SNMP
- APIs : APIs spécialisés pour systèmes avancés
- ...

Dans la mailing list de ReactOS¹⁶, on peut voir une discussion récente à propos de l'implémentation de lwIP dans ReactOS comme étant stable et devenue la stack primaire du système, cependant durant tous mes tests, je n'ai jamais vu cette partie de code être utilisée (leur stack IP par contre bien). Seule la partie TCP du système utilise lwIP en réalité. Etant en layer 4, ce point n'a pas impacté mon développement au final.

15. <https://savannah.nongnu.org/projects/lwip/>

16. <https://lists.gnu.org/archive/html/lwip-devel/2011-08/msg00010.html>

FIGURE 28 – Couche réseau et utilisation de lwIP



11.4 Syntaxe de la commande unshare.exe

Listing 19 – Utilisation de unshare.exe

```
unshare // Lance un cmd.exe dans un nouveau namespace
unshare attach <id> // Ajoute une veth liée à l'interface <id>
unshare detach <id> // Détache une interface virtuelle (déchargement)
unshare ip <id> <ipv4> <mask> // Attribue une adresse IPv4 à une interface
unshare mac <id> <mac> // Change l'adresse MAC d'une interface
unshare enable switch // Active le switch layer 2 virtuel
unshare disable switch // Désactive le switch layer 2 virtuel
```

11.5 Blocage d'implémentation par le Plug'n'Play manager et NDIS Miniport

Jusqu'à présent, tout le système a été basé pour attacher une interface virtuelle à une interface physique¹⁷. Pour bien faire, il faudrait l'attacher à une interface de type TAP plutôt.

11.5.1 Interfaces type TAP

Les interfaces TAP sont des interfaces layer 2 dont la fonction n'est que de forwarder le trafic du kernel-mode en user-mode. Ces interfaces sont utilisées par OpenVPN ou encore le réseau TOR. En effet, la partie intéressante du code se trouve en user-mode (le chiffrement, la négociation de connexions, la gestion du trafic, le décodage des trames, etc.). L'interface TAP est très simple vu que le driver ne s'occupe que de faire du passthrough entre le kernel et le user-mode.

L'avantage d'utiliser ce type d'interface est qu'on dispose d'une carte réseau virtuelle layer 2 sans dépendance hardware. Il existe un portage pour Windows de l'interface TAP et elle semble fonctionner dans ReactOS, mais le système ne gère pas pleinement le Plug'n'Play à ce niveau là. En effet, pour pouvoir installer et instancier une interface TAP dans ReactOS, un reboot du système est nécessaire... autant dire que ce n'est absolument pas envisageable pour les namespaces qui sont totalement dynamiques et, qui plus est dans mon implémentation, volatiles.

J'ai passé beaucoup (trop) de temps sur la tentative de faire fonctionner l'interface TAP pour pouvoir l'utiliser comme pièce maîtresse du mécanisme d'interface virtuelle.

17. Concrètement, il faut une interface « physique » (pour ReactOS) par Virtual LAN que l'ont veut créer.

11.5.2 Interfaces virtuelles NDIS

Les interfaces virtuelles dans NDIS sont des pilotes Miniport qui ne font pas d'appels au hardware (pas de handlers d'interruption, etc.). Tout s'enregistre de la même façon, excepté que l'appel à l'instanciation du driver ne se fait pas quand le hardware est détecté (vu qu'il n'y en a pas) mais bien manuellement ou automatiquement quand NDIS est prêt de son côté.

Lors de l'enregistrement d'une interface Miniport, la structure de configuration contient un champ `InitializeHandler`. Le problème actuellement dans l'implémentation de NDIS est que ce handler n'est appelé **que** lors d'un appel du Plug'n'Play manager « hardware ». Du coup l'interface TAP (ou n'importe quelle interface sans correspondance hardware) n'est jamais instanciée (à noter que dans le gestionnaire de périphériques, l'interface apparaît bien et est notée comme fonctionnelle (mais encore une fois, cette partie n'est sans doute pas bien implémentée)).

11.6 Modifications collatérales

Durant le développement général, je suis tombé plusieurs fois sur des cas qui ne sont pas directement liés à l'implémentation des namespaces. Certains cas m'ont bloqué, d'autres sont totalement inutiles mais intéressants d'un point de vue technique.

11.6.1 Mode Promiscuous dans le driver pcnet

En regardant le code du driver pcnet, le mode promiscuous est bien défini dans le header du driver (`/drivers/network/dd/pcnet/pcnethw.h`) mais le define `CSR15_PROM` n'est utilisé nulle part dans le code. Du coup, j'ai forcé son utilisation durant l'initialisation de l'interface, cela se passe dans la fonction `MiInitChip` dans `/drivers/network/dd/pcnet/pcnet.c` ligne 742.

Listing 20 – `/drivers/network/dd/pcnet/`

```
//
// pcnethw.h
//
#define CSR15_DRX      0x1      /* disable receiver */
#define CSR15_DTX      0x2      /* disable transmitter */
#define CSR15_LOOP     0x4      /* loopback enable */
// [...]
#define CSR15_DRCVBC   0x4000   /* disable receive broadcast */
#define CSR15_PROM     0x8000   /* promiscuous mode */

//
// pcnet.c MiInitChip(): le CSR15 étant initialisé à 0x00
//
NdisRawWritePortUshort(Adapter->PortOffset + RAP, CSR15);
NdisRawWritePortUshort(Adapter->PortOffset + RDP, CSR15_PROM);
```

11.6.2 Changer la couleur du Blue Screen

Quand on développe dans le kernel, on fini toujours par tomber au moins une (ou 10... ou 50...) fois sur un blue screen parce qu'on a déréférencé un pointeur NULL (ou qu'on a essayé de charger une font TrueType dans le kernel par exemple). A force, le bleu du BSOD ça lasse un peu. Puis, n'importe quel geek a toujours voulu au moins une fois dans sa vie changer la couleur de fond d'un Blue Screen, c'est tellement fun.

On a l'avantage ici d'avoir le code source du kernel, alors ça doit être facile de changer ça. Suffit de chercher un peu comment le Blue Screen est généré. Dans le debugger kernel intégré au système, une commande permet de tester si l'équivalent du `panic()` sous Linux fonctionne. C'est `bugcheck`. Cette commande génère un Blue Screen tout simplement. De là, facile de savoir où le code se trouve.

L'affichage du Blue Screen se trouve dans `/ntoskrnl/ke/bug.c`, j'ai dû parcourir le fichier et lire un peu ce qui s'y trouve pour tomber sur un appel à `InbvSolidColorFill`. En changeant un peu les valeurs en paramètres aléatoirement, j'ai fini par trouver que le dernier paramètre est la couleur de fond. J'ai changé le `0x02` en `0x04` et je me suis retrouvé avec un Blue Screen... vert.

FIGURE 29 – Green Screen of Death



```
A problem has been detected and ReactOS has been shut down to prevent damage
to your computer.

MANUALLY_INITIATED_CRASH

If this is the first time you've seen this stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any ReactOS updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup options, and then
select Safe Mode.

Technical information:
*** STOP: 0x000000E2 (0x00000000,0x00000000,0x00000000,0x00000000)
```

11.7 Comparaison des interfaces graphiques de gestion des protocoles

Comme dit dans le développement, la gestion (activation/désactivation) d'un ProtocolDriver n'est pas finie et l'interface graphique (par exemple) de ReactOS manque cruellement de possibilités.

FIGURE 30 – VirtualBox Bridged Networking Driver (NDIS ProtocolDriver)

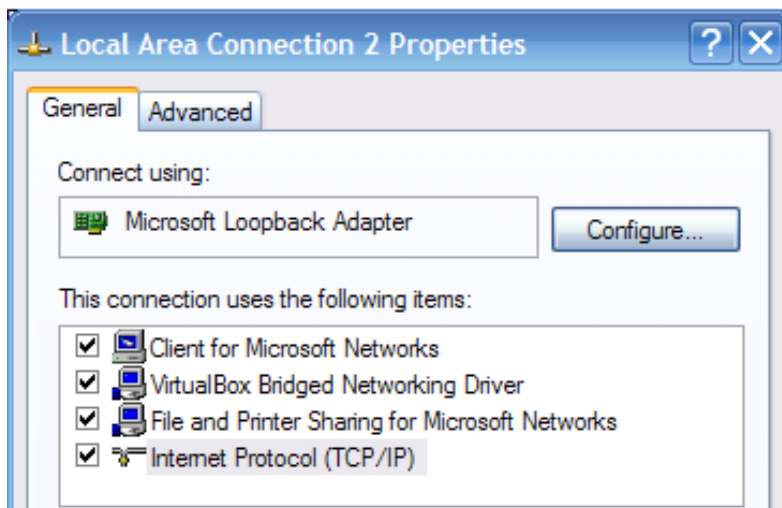


FIGURE 31 – Interface graphique non finalisée dans ReactOS

