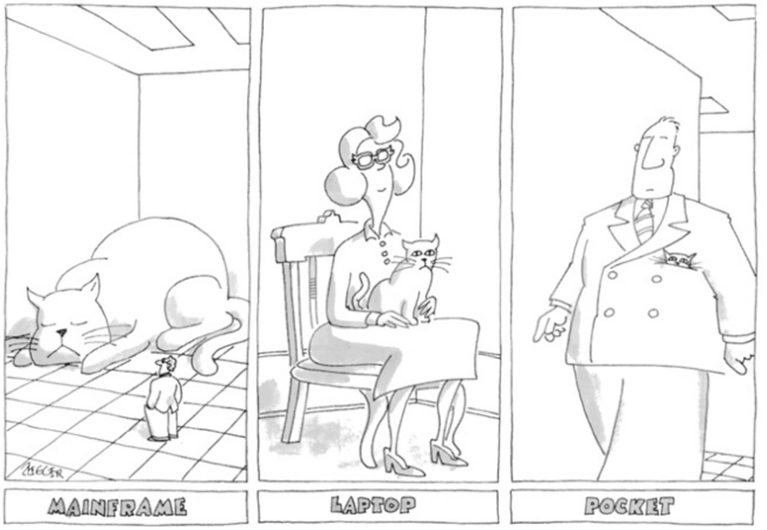
Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année



UE Réseaux Avancés II — P-F. Bonnefoi

Version du 3 mai 2021

Table des matières

Plan de la partie sécurité

- ▷ L'«*OS fingerprint*» ou la reconnaissance d'un système grâce à sa pile réseau ;
- ▷ Les risques d'attaques réseau et des réponses protocolaires ;
- ▷ Diminuer la latence : «*TCP Fast-open*» et la gestion du risque.

D'après Wikipedia

La prise d'empreinte de la pile TCP/IP (en anglais : «*TCP/IP stack fingerprintin*»g ou «*OS fingerprinting*») est un procédé permettant de déterminer l'identité du système d'exploitation utilisé sur une machine distante en analysant les paquets provenant de cet hôte.

Il y a deux types différents de fingerprinting :

- ⋆ L'**OS fingerprinting passif** : identifier le système d'exploitation uniquement à l'aide des paquets qu'il reçoit ;
- ⋆ L'**OS fingerprinting actif** : envoie des paquets et attend les réponses (ou l'absence de réponses). Ces paquets sont parfois formés de **manière étrange** car les différentes implémentations des piles TCP/IP répondent différemment face à de telles erreurs.

Quelques outils de détection d'OS :

- ▷ Actif : Nmap, xprobe [1], Scapy,
- ▷ Passif : p0f (basé sur ouvertures de connexion TCP, paquets TCP SYN), Scapy, Ettercap, le pare-feu Netfilter de noyau >2.6.31 de Linux.

Comment faire de l'OS fingerprint ?

- ⋆ en utilisant les protocoles applicatif (IMAP, POP, SMTP, SSH, FTP *etc.*) :
  - ◇ le «*banner grabbing*» : en se connectant au service et en récupérant la bannière d'accueil qui contient des infos sur l'hôte. *Le principe du «Security by Obscurity» : changer les bannières pour dissimuler le système* ;
  - ◇ le «banner grabbing» passif : dans l'en-tête d'un courrier, dans celui de la réponse du serveur HTTP, dans la récupération d'une commande du système au travers d'un FTP pour voir comment elle a été compilé *etc*
- ⋆ en utilisant les **particularités** des protocoles TCP :
  - ◇ les réponses à des paquets inattendus, le temps de réponse, les valeurs des options *etc.*
- ⋆ en utilisant le protocole SNMP, «*Simple Network Management*» : s'il est installé et configuré par défaut, il peut informer sur la nature du matériel ;
- ⋆ en identifiant les ports ouverts pour détecter les services présents sur l'hôte :
  - ◇ le balayage de ports, «*port scanning*» : passer en revue les différents ports pour trouver ceux ouverts, c-à-d où un serveur attend (listen) : windows : port 137, netbios-ns, NetBIOS Name Service, le mDNS *etc.*

Méthodologie

Pour effectuer un «*fingerprinting*», on mesure des **comportements** de la machine cible que l'on **compare** avec des comportements pré-enregistrés dans une BD : la comparaison de plusieurs comportements (valeurs prises, temps mis pour répondre *etc.*) permet **d'identifier** avec plus ou moins de précision la cible.

«Port scanning» : trouver les ports ouverts, fermés ou filtrés

Pour le comportement de la pile TCP/IP, **au niveau TCP** :

- ⋆ envoyer un segment avec le bit FIN : d'après la RFC 793, la réception d'un segment FIN sans connexion préalable doit être ignorée, mais certaines piles TCP/IP renvoient un RST $\Longrightarrow$ fingerprint ;
- ⋆ «*TCP ISN sampling*» : tester l'évolution des numéros de séquence lors de l'établissement de connexion TCP et déterminer comment il évolue : de manière incrémentale, par rapport à une horloge, de manière aléatoire *etc.*
- ⋆ les options TCP : *quelles sont les options présentes, leur valeur, leur ordre dans le segment ?*
- ⋆ le Timestamp de TCP : *comment évolue-t-til ?*
- ⋆ le timeout de TCP : envoyer un simple SYN, ignorer le SYN/ACK et mesurer le temps pris pour l'hôte pour réenvoyer ce SYN/ACK (à l'aide de l'option TimeStamp on évite les erreurs dues aux délais du réseau).
  Cette mesure est très efficace et est mise en œuvre par l'outil RING, «Remote Identification Next Generation»*.*

Pour le comportement de la pile TCP/IP, **au niveau IP** :

- ◇ l'IPID, ou l'identifiant du datagramme IP : souvent à zéro mais pouvant valoir une valeur aléatoire ou incrémenter de un à chaque transmission ;
- ◇ la gestion de la fragmentation : *est-ce qu'un fragment qui «overlap», c-à-d recouvre, un segment précédent est pris en compte ou non ?*

Pour le comportement de la pile TCP/IP, **au niveau ICMP** :

- ⋆ leur TOS (différent de zéro sous GNU/Linux),
- ⋆ leur contenu dans le cas d'erreur :
  - ◇ une partie du datagramme IP qui a entraîné l'erreur «*ICMP Error Quoting*» ;
  - ◇ le message ICMP reçu dans le cas d'une erreur dans ce message ICMP «*ICMP Error Message Echo Integrity*» ;

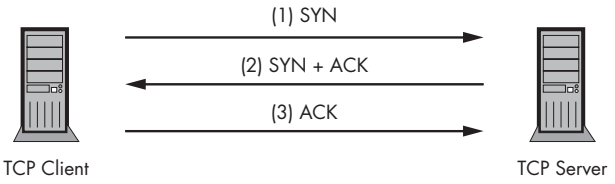
- **déterminer si la cible est sur le réseau** : envoie un ICMP ping request, et essaye de se connecter sur le port 80 (HTTP) ;
- **balayer les ports ouverts** pour trouver au moins un port ouvert (une application est en écoute et attend une connexion) et un port fermé (pas d'application attachée à ce port) ;
- déclencher l'**OS fingerprint** :
  - envoyer un paquet unique SYN : initier une connexion TCP ;
  - envoyer un second paquet sans drapeaux (pas de SYN, ACK, PSH *etc.*). *Ce paquet est appelé «null scan»*
  - envoyer un troisième paquet avec les drapeaux : URG, PSH, SYN et FIN
    Ces paquets sont appelées dans la RFC 1025 : nastygram, kamikaze, christmas tree
  - en cas de service connu ouvert : faire du «*banner grabbing*» :

darkstar

```
pef@darkstar:~/$ nmap -A 192.168.42.83
Starting Nmap 5.51 ( http://nmap.org ) at 2011-10-09 17:25 CEST
Nmap scan report for 192.168.42.83
Host is up (0.0023s latency).
Not shown: 996 closed ports
PORT     STATE    SERVICE    VERSION
22/tcp   open     ssh        OpenSSH 5.8p1 Debian 1ubuntu3 (protocol 2.0)
| ssh-hostkey: 1024 8c:02:1e:99:51:e6:29:21:46:95:6e:95:a5:ed:b7:50 (DSA)
|_2048 cc:1b:0d:e8:36:65:fe:ac:61:10:50:35:98:18:81:f9 (RSA)
53/tcp   open     domain     dnsmasq 2.57
80/tcp   open     http       Apache httpd 2.2.17 ((Ubuntu))
|_http-title: Site doesn't have a title (text/html).
8080/tcp filtered http-proxy
Service Info: OS: Linux

Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 22.11 seconds
```


- les options de nmap :
  - TCP connect() scan—(Nmap -sT) : envoi un SYN et ACK (connexion normale) ;



darkstar
```
pef@darkstar:~$ nmap -sT -p80 192.168.42.83
```

Ici, le port ciblé est le port `80`.

darkstar
```
pef@darkstar:~$ nmap -sT -p80 192.168.42.83
Starting Nmap 5.51 ( http://nmap.org ) at 2011-10-09 18:30 CEST
Nmap scan report for 192.168.42.83
Host is up (0.0016s latency).
PORT   STATE SERVICE
80/tcp open  http
```

La trace recueillie avec `tcpdump` :

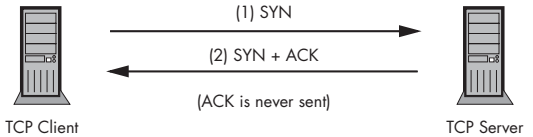
solaris
```
pef@solaris:~$ sudo tcpdump -i eth0 -nvvX host 192.168.42.122 and host 192.168.42.83
18:26:35.987561 IP (tos 0x0, ttl 255, id 33680, offset 0, flags [DF], proto TCP (6), length 64)
    192.168.42.122.56733 > 192.168.42.83.80: Flags [S], cksum 0xb01e, seq 604272313, win 65535,
    options [mss 1460,nop,wscale 3,nop,nop,TS val 71119834 ecr 0,sackOK,eol], length 0

18:26:35.987666 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 56)
    192.168.42.83.80 > 192.168.42.122.56733: Flags [S.], cksum 0xd648, seq 832218517, ack
604272314, win 14480,
    options [mss 1460,sackOK,TS val 212522325 ecr 71119834], length 0

18:26:35.987892 IP (tos 0x0, ttl 255, id 37505, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.42.122.56733 > 192.168.42.83.80: Flags [.], cksum 0x36ab, seq 1, ack 1, win 65535,
    options [nop,nop,TS val 71119834 ecr 212522325], length 0

18:26:35.987900 IP (tos 0x0, ttl 255, id 48576, offset 0, flags [DF], proto TCP (6), length 40)
    192.168.42.122.56733 > 192.168.42.83.80: Flags [R.], cksum 0x89d5, seq 1, ack 1, win 65535,
length 0
```

- TCP SYN or half-open scan—(Nmap -sS) : envoi un seul SYN (pas de log sur la cible, mais attention à la réception d'un SYN/ACK ou RST/ACK) ;



darkstar
```
pef@darkstar:~$ sudo nmap -sS -p80 192.168.42.83
```

Ici, le port ciblé est le port `80`*.*

darkstar
```
pef@darkstar:~$ sudo nmap -sS -p80 192.168.42.83
Starting Nmap 5.51 ( http://nmap.org ) at 2011-10-09 18:39 CEST
Nmap scan report for 192.168.42.83
Host is up.
PORT   STATE    SERVICE
80/tcp filtered http

Nmap done: 1 IP address (1 host up) scanned in 15.06 seconds
```

La trace recueillie avec `tcpdump` :

solaris
```
pef@solaris:~$ sudo tcpdump -i eth0 -nvvX host 192.168.42.122 and host 192.168.42.83
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
18:40:08.795967 IP (tos 0x0, ttl 42, id 64924, offset 0, flags [none], proto TCP (6), length 44)
    192.168.42.122.62082 > 192.168.42.83.80: Flags [S], cksum 0x50d4, seq 1064973031, win 3072,
    options [mss 1460], length 0

18:40:08.796043 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 44)
    192.168.42.83.80 > 192.168.42.122.62082: Flags [S.], cksum 0xd63c, seq 707071997, ack
1064973032, win 14600,
    options [mss 1460], length 0

18:40:08.856037 IP (tos 0x0, ttl 64, id 22009, offset 0, flags [DF], proto TCP (6), length 40)
    192.168.42.122.62082 > 192.168.42.83.80: Flags [R], cksum 0xadbb, seq 1064973032, win 0,
length 0
```

NMap renvoie un RST après la réception du SYN/ACK du serveur comme si la connexion n'avait pas été demandée.

Un outil d'audit : nmap

- TCP FIN, XMAS, and NULL scans—(Nmap -sF, -sX, -sN) : envoi de segments avec les drapeaux choisis ;

darkstar

```
pef@darkstar:~$ sudo nmap -sF -p80 192.168.42.83
Starting Nmap 5.51 ( http://nmap.org ) at 2011-10-09 18:54 CEST
Nmap scan report for 192.168.42.83
Host is up.
PORT   STATE         SERVICE
80/tcp open|filtered http
```

solaris

```
pef@solaris:~$ sudo tcpdump -i eth0 -nvvX host 192.168.42.122 and host 192.168.42.83
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
18:54:43.662791 IP (tos 0x0, ttl 53, id 25131, offset 0, flags [none], proto TCP (6), length 40)
    192.168.42.122.47432 > 192.168.42.83.80: Flags [F], cksum 0xcc21, seq 1638066792, win 2048,
length 0

18:54:44.662309 IP (tos 0x0, ttl 43, id 33275, offset 0, flags [none], proto TCP (6), length 40)
    192.168.42.122.47433 > 192.168.42.83.80: Flags [F], cksum 0xc41e, seq 1638132329, win 4096,
length 0
```

- TCP ACK scan—(Nmap -sA) : envoi un ACK et attend un RST en retour ;

darkstar

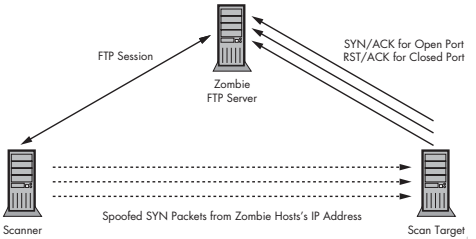
```
pef@darkstar:~$ sudo nmap -P0 -e en0 -sA -p80 192.168.42.83
Starting Nmap 5.51 ( http://nmap.org ) at 2011-10-09 19:03 CEST
Nmap scan report for 192.168.42.83
Host is up.
PORT   STATE    SERVICE
80/tcp filtered http
```

solaris

```
pef@solaris:~$ sudo tcpdump -i eth0 -nvvX host 192.168.42.122 and host 192.168.42.83
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
19:03:45.471095 IP (tos 0x0, ttl 39, id 55545, offset 0, flags [none], proto TCP (6), length 40)
    192.168.42.122.55365 > 192.168.42.83.80: Flags [.], cksum 0xba54 (correct), seq 0, ack
3429526113, win 4096, length 0

19:03:45.472226 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40)
    192.168.42.83.80 > 192.168.42.122.55365: Flags [R], cksum 0xca60 (correct), seq 3429526113,
win 0, length 0
```

- ◇ TCP idle scan—(Nmap -sI) :



  - ⋆ on se sert d'un intermédiaire sur lequel on est connecté en TCP ;
  - ⋆ on spoofe l'@IP de l'intermédiaire pour scanner la cible ;
  - ⋆ on surveille les IP ID de la connexion avec l'intermédiaire pour voir s'ils sont modifiés (l'intermédiaire envoie un RST à la cible et incrémente cet IP ID).

- ◇ UDP scan—(Nmap -sU) : envoi d'un datagramme UDP et attente d'un ICMP de type «Port unreachable» pour savoir si le port est ouvert ou fermé.

▷ les valeurs retournées par le «*scannage*» de ports sont :

- ◇ «*open*» : il y a une application accessible associée à ce port ;
- ◇ «*closed*» : il n'y a pas d'application attachée ;
  On peut le simuler à l'aide d'une règle du firewall NetFilter :

```
xterm
$ sudo iptables -t filter -A INPUT -p tcp --dport 9000 -j REJECT --reject-with tcp-reset
```

- ◇ «*filtered*» : il peut y avoir une application attachée à ce port, mais les communication sont bloquées.
  Il n'y a pas de «RST» envoyé, en le bloquant avec le firewall «netfilter» sous GNU/Linux :

```
xterm
$ sudo iptables -A OUTPUT -p tcp --sport 80 --tcp-flags RST RST -j DROP
```

Les règles du RST : RST ou RST/ACK ?

- ⋆ à la réception d'un SYN sur un port fermé on renvoie un RST/ACK ;
- ⋆ à la réception d'un SYN/ACK sur un port fermé, on renvoie un RST.

Les risques d'attaques dans un réseau (quelques uns)

Plan

- Les attaques en «déni de service», *DoS*, sur la pile TCP/IP et la parade avec le «*SYN cookie*» ;
- Les attaques par fragments et leur traitement par le Firewall ;
- Quelques attaques «communes» sur le fonctionnement d'une machine dans un réseau local. *Et malheureusement, toujours efficaces…*

Terminer un connexion qui ne nous appartient pas

La RFC 793 concernant TCP dit qu'un paquet RST est valide si son numéro de séquence TCP se trouve dans la fenêtre TCP. Il est possible de transmettre un segment TCP demandant l'interruption d'une connexion TCP en cours en trouvant un bon numéro de séquence.

Usurper l'identité d'une machine et communiquer à sa place : «*IP spoofing*»

- Les **machines en présence** :
  - A machine dont l'adresse est usurpée ;
  - B machine sur laquelle on veut se connecter en se faisant passer pour A ;
  - C machine de l'attaquant ;
- Le **déroulement de l'attaque** :
  - C envoi une **demande de connexion** (SYN) en utilisant comme source l'adresse de A ;
  - B renvoi alors un **SYN/ACK** avec la synchronisation du numéro de séquence choisi par C et son propre numéro de séquence à synchroniser ;
  - A reçoit l'envoi de B :
    - cette envoi ne correspondant à rien elle renvoi un RST dès que possible ;
    - **Mais** A est bloquée par C qui la sature de demande de connexion (**SYN flood**) ;
      Le fait que la pile TCP/IP de A soit bloquée est appelé un déni de service, ou DoS.
    - $\Longrightarrow$ B ne reçoit pas de paquet de la part de A ;
  - C envoi un paquet à B en **essayant de trouver** le numéro de séquence envoyé par B à A ;
  - Si C arrive à trouver le **bon numéro de séquence**, alors il peut envoyer des données à B en faisant croire qu'il est A.

Cela marche pour de nombreux protocoles où les réponses du serveur ne sont pas importantes.

Le SYN Cookie

C'est une méthode pour éviter les DoS, *Denial of Service*, par épuisement de ressource sur l'établissement des connexions TCP avec l'attaque «*SYN flood*».

Fonctionnement normal de l'établissement d'une connexion :

a. réception d'une demande de connexion (segment TCP avec SYN) ;
b. réservation d'un **espace mémoire** pour gérer la connexion ;
c. une fois la connexion établie cette mémoire est utilisée pour gérer la communication (TSAP, taille de fenêtre, valeur de synchronisation, …) ;
d. à la fin de la communication, la connexion est terminée et cette mémoire est libérée.

Attaque par Syn Flood :

i. de nombreux SYN sont reçus (inondation ou *flooding*) ;
ii. un espace mémoire est réservé pour chacun ;
iii. le système répond par un SYN/ACK et attend le ACK correspondant qui ne viendra jamais...
iv. **épuisement des ressources !**

Solution :

▷ **retarder l'établissement** de la connexion (réservation de mémoire) jusqu'à l'établissement effective de la connexion (réception du ACK en retour de la part du client).

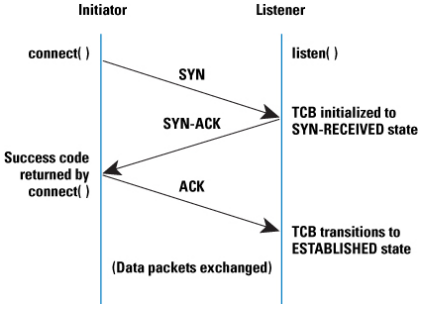
Comment ? Le SYN cookie !

Reprend l'idée du Cookie du protocole HTTP gérer par le navigateur Web :

1. demander au client de **conserver les données** de connexion ;
2. le client se connecte, le serveur lui **donne des infos** et ne réserve pas de ressources ;
3. si le client revient il **redonne le cookie** et le serveur réserve de la mémoire pour la connexion.

Mais cela doit rester compatible avec le fonctionnement normal de TCP !

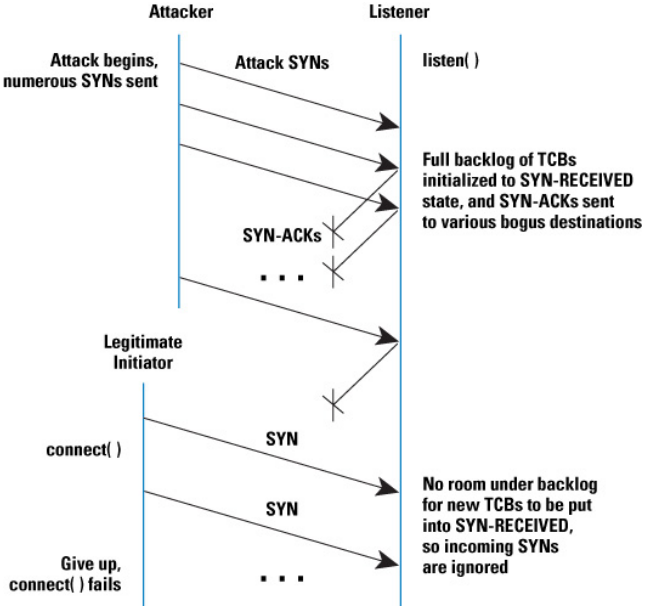
TCP on connait



À chaque demande de connexion, un «TCB», *transmission Control Block*, est alloué pour gérer la connexion (changement d'état, TSAP source et destination).

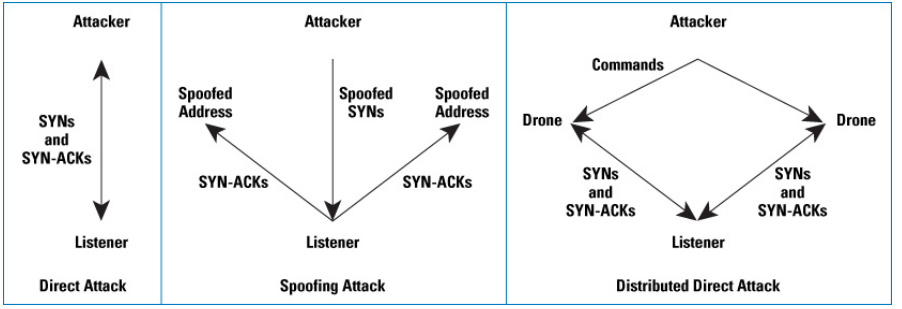
Déroulement de l'attaque :

a. envois de SYN en provenance d'adresses spoofées
b. épuisement de la table des TCB...
c. blocage des utilisateurs légitimes !



Le SYN Cookie

Une attaque difficile à contrer avec un firewall

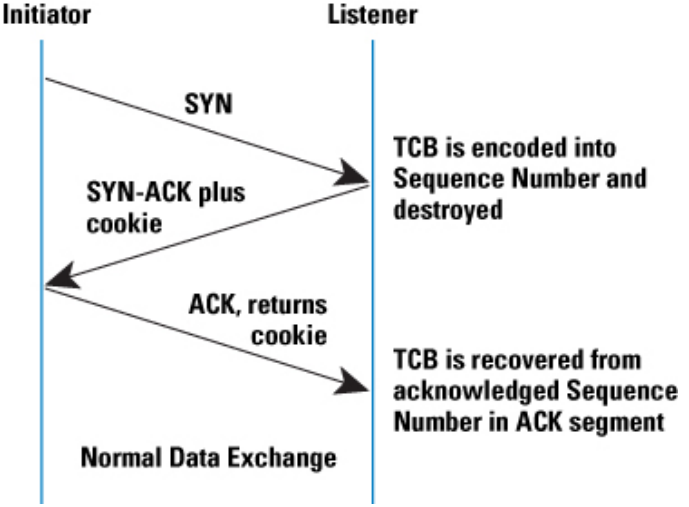


Comment filtrer suivant l'origine du datagramme ?

La solution :

- s'inspirer du cookie HTTP ;
- *Où mettre le cookie ?*

  Dans le numéro de séquence !

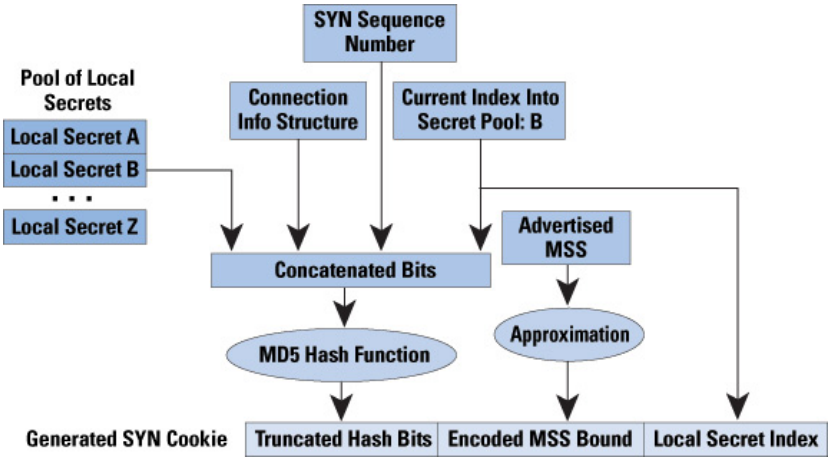


Le SYN Cookie : fabrication

Fabrication du cookie

On utilise des données diverses, dont certaines non prévisibles, et une **fonction de hashage** :

- un secret local qui varie : un compteur évoluant avec le temps.
  Ainsi, si le cookie semble venir d'un « temps » trop lointain on l'ignore.
- le TSAP source et destination pour les info de la structure de connexion ;
- le numéro de séquence du client ;
- le rang du compteur (la valeur du compteur modulo la taille de la représentation sur 5 bits) ;
- un hash est calculé, on en conserve 3 octets (protection : 2^{24}) ;
- il reste 8 bits : 3 bits pour un MSS (8 possibilités seulement) + 5 bits pour le rappel du rang du compteur (32 valeurs).



Problèmes ? Valeur approchée du MSS, perte des extensions présentes dans le segment TCP initial (négociation de la taille de la fenêtre, du SACK, *etc.*)

Rappel sur la fragmentation

- ◇ tous les fragments issus d'un même datagramme possèdent le même identifiant, IP ID, sur 16bits ;
- ◇ réalisé par la couche 3 (couche réseau ou IP), sans tenir compte de la couche 4 (couche transport) ;
- ◇ le destinataire ne peut connaître le nombre de fragments à l'avance ;
- ◇ la perte d'un fragment force à détruire le reste du datagramme reçu (utilisation d'un timer) ;
- ◇ la couche IP étant sans contrôle d'erreur il n'y a pas de tentative de correction de cette perte.

Présentation des risques

La fragmentation des datagrammes IP permet de **masquer** des segments TCP aux outils de filtrages utilisés dans les routeurs et dans les machines.
On parle «d'évasion».

Travail du firewall

Le fonctionnement du firewall consiste à appliquer les règles de filtrage sur le premier fragment et de mémoriser le résultat de cette application :

- ⋆ le module de filtrage mémorise une liste de paquets indexés par (@src, @dst, proto, IP ID) ;
- ⋆ lorsque le premier fragment est traité avec le bit MF positionné à 1, un enregistrement est créé ;
- ⋆ lorsque les fragments suivants arrivent, cet enregistrement est utilisé pour filtrer ou non ces fragments.

Il est possible que des fragments arrivent dans le désordre, et que l'on ne puisse pas traiter le premier fragment pour éventuellement rejeter l'ensemble de ces fragments.
Dans ce cas là, les fragments arrivent sur la machine cible et il n'y a pas de problème dans la mesure où sans ce premier fragment il ne sera pas possible de reconstituer le paquet complet, en cas de filtrage de celui-ci.

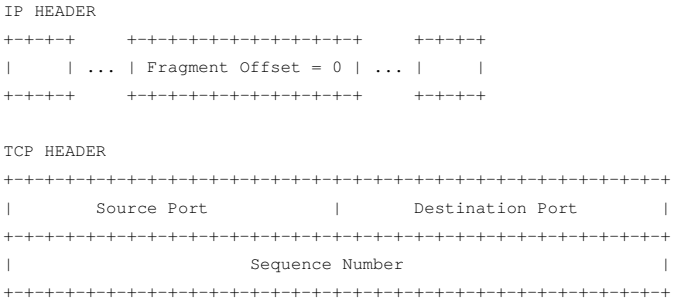
Tiny Fragment Attack

Cette attaque utilise la taille minimale de fragment.

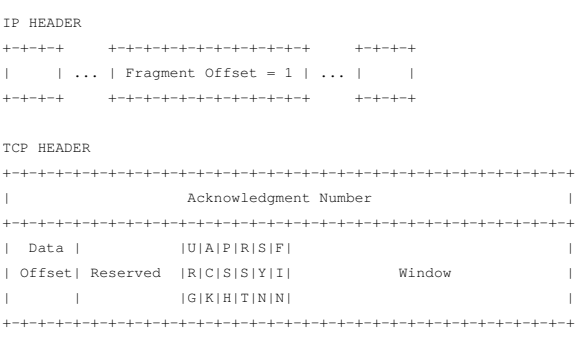
Si la taille du fragment est **suffisamment petite**, il est possible de forcer une partie de l'en-tête du segment TCP dans le second fragment.

Cela permet d'éviter que les règles du firewall s'appliquent sur cette en-tête.

Fragment 1 : inoffensif, il ne contient aucune information sur les drapeaux (SYN, ACK *etc.*) du segment TCP.



Fragment 2 : contient les fameux drapeaux…



Exemple

Dans le cas où les demandes de connexion (SYN=1 et ACK=0) sont rejetées par leur firewall, on utilise un **fragment de petite taille** pour dissimuler cette demande de connexion : le fragment ne contiendra pas les informations de connexions TCP correspondant aux «*flags*».

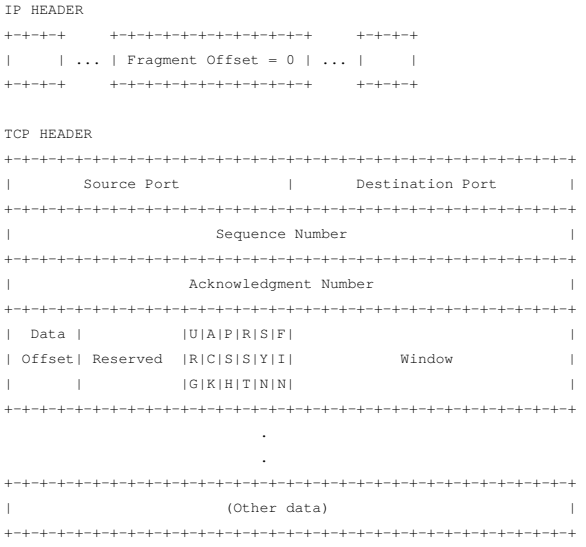
Overlapping fragment attack

D'après la RFC 791, l'algorithme de réassemblage des fragments permet à de nouveaux fragments de

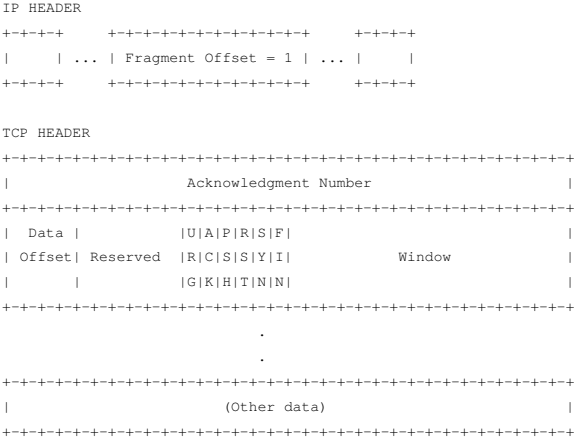
Un attaquant peut alors construire une suite de fragments où :

- ▷ le fragment d'offset le plus petit (à 0) contient des données non dangereuses et ainsi va pouvoir traverser le firewall ;
- ▷ les fragments suivants contenant un offset supérieur à 0, vont réécrire une partie de l'en-tête du paquet TCP déjà reçu (comme le port destination). **Ces fragments ne devraient pas alors être filtrés par le firewall.**

Fragment 1 : valeurs acceptables pour le firewall : (SYN=0, ACK=1)



Fragment 2 : avec un offset de 8 octets, va réécrire les indicateurs en SYN=1, ACK=0



Tiny Overlapping Fragment Attack

- Fragment 1 : (fragment offset = 0 ; length ≥ 16)
- Fragment 2 : (fragment offset = 0 ; length = 8)
- Fragment 3 : (fragment offset ≥ 2 ; length = reste du message)

Les règles du firewall

No	Action	Source Port	Dest. Port	Flags	Purpose
1	Permit	>1023	SMTP	ANY	Incoming E-mail
2	Permit	>1023	ANY	Ack=1	Existing FTP data channel connections.
3	Deny	ANY	ANY	ANY	Default deny

Tiny Overlapping Fragment Attack

- Fragment 1 : (fragment offset = 0 ; length ≥ 16)
  `Attaquant(1234) ? Cible(SMTP) SYN = 0 ; ACK = 0`
- Fragment 2 : (fragment offset = 0 ; length = 8)
  `Attaquant(1234) ? Cible(Telnet)`
- Fragment 3 : (fragment offset ≥ 2 ; length = reste du message)

Les règles du firewall

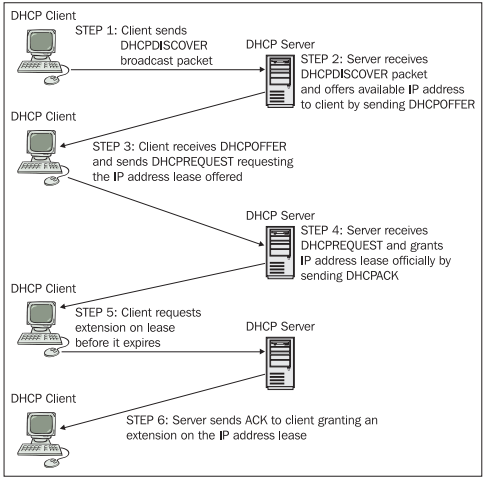
No	Action	Source Port	Dest. Port	Flags	Purpose
1	Permit	>1023	SMTP	ANY	Incoming E-mail
2	Permit	>1023	ANY	Ack=1	Existing FTP data channel connections.
3	Deny	ANY	ANY	ANY	Default deny

- sur **la couche 1**, la couche «physique» :
  - coupage de câble, de fibre optique ;
  - introduction de courant à haut voltage dans les câbles ;
  - bloquer les connexions sans-fil (utilisation d'un micro-onde ?)
  - perturbation des câbles (génération d'ondes électromagnétiques à proximité des câbles filaires).

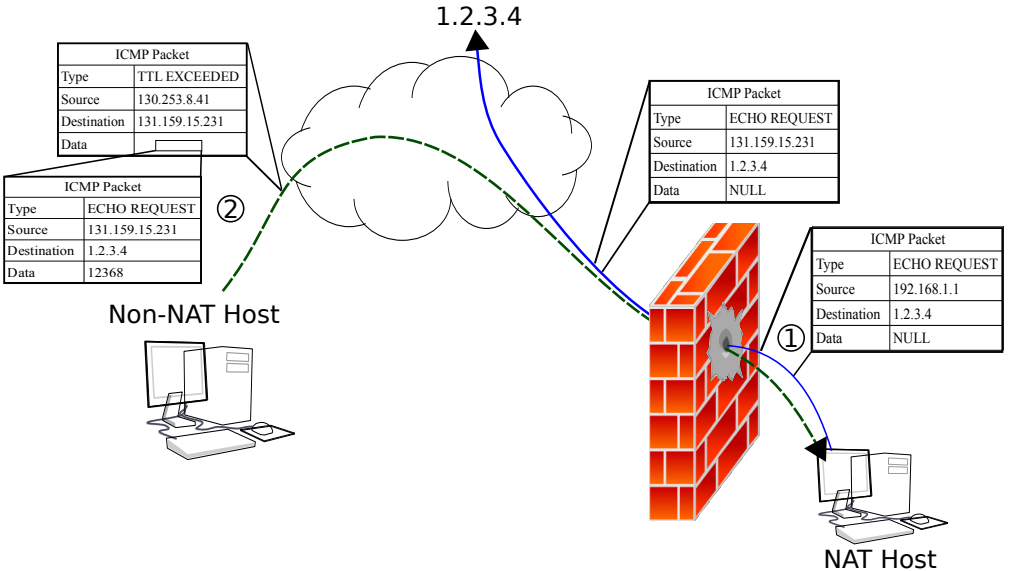
- sur **la couche 2**, la couche «liaison de données» :
  - les **attaques sur le MAC** :
    - attaque sur la CAM, «*Content Addressable Memory*» d'un switch :
      - cette table sert à mémoriser les @MAC des matériels connectés à chaque port d'un switch, ainsi que leur appartenance éventuelle à un VLAN particulier ;
      - l'attaque vise à saturer cette table et à «dissuader» le «*switch*» de l'utiliser : il fonctionne alors comme un «hub» et envoie les trames sur tous les ports du switch : il est alors possible d'écouter les paquets qui ne nous sont pas destinés !

    - l'**ARP Spoofing** :
      - consiste à prendre l'identité d'un autre matériel (peut être bloqué si le switch si celui-ci en est capable).
      - on peut prendre l'**identité du routeur** pour intercepter l'intégralité du trafic .
        On parlera d'attaque «Man-In-the-Middle» et d'empoisonnement de cache ARP, «cache poisoning» *pour la victime*.

- ◇ les **attaques sur le DHCP** (IPv4 : RFC 2131, IPv6 : RFC 3315) :
  - ⋆ fonctionnement du DHCP :
    - ▷ ce protocole sert à fournir une configuration réseau à un poste qui en fait la demande à un serveur : @IP, masque de sous-réseau, routeur par défaut, adresse des serveurs DNS, *etc*.
    - ▷ le client «*broadcast*» une demande de prise en charge par un serveur DHCP (DHCPDISCOVER) ;
      IPv4 : paquet UDP, port destination 67, port source 68, @IP dest `255.255.255.255` *et @IP source* `0.0.0.0`.
      IPv6 : paquet UDP, port destination 547, port source 546, @IP destination `ff05::1:3`
    - ▷ si un serveur DHCP est présent dans le réseau, il peut répondre à la requête du client (DHCPOFFER) ;
    - ▷ l'@IP est «louée» par le serveur à un client pour une certaine durée, «*leasing*» : lorsque l'attribution va expirer, le client doit demander le renouvellement de la période.



  - ⋆ Une **première attaque** consiste à **épuiser les @IP** disponibles sur le serveur
    - ▷ l'attaquant exécute un grand nombre de demandes de prise en charge par le serveur DHCP.
    - ▷ les clients légitimes ne peuvent plus obtenir d'@IP.
  - ⋆ une **seconde attaque** consiste à **remplacer le serveur DHCP**, on parle de «rogue DHCP server» :
    - ▷ le client accepte le premier DHCPOFFER reçu ;
    - ▷ le faux serveur DHCP fournit, par exemple, substitue son adresse à celle du routeur pour les clients.
      La RFC 3118 propose une extension du protocole DHCP : utiliser un HMAC et une clé différente pour chaque client connue également du serveur : permet d'authentifier les messages échangés.



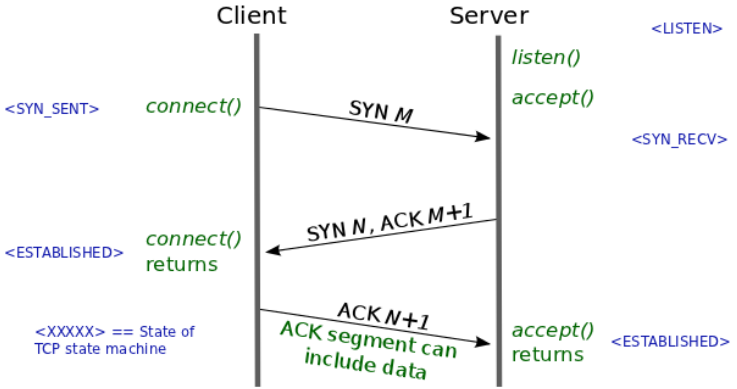
But : *atteindre une machine derrière un NAT depuis l'extérieur* ⟹ *faire un trou dans le firewall !*

▷ la machine extérieure, «*Non-NAT Host*» connait l'adresse public de la machine intérieur, «*NAT Host*» ;

▷ la machine intérieure envoie du trafic icmp vers une adresse IP inexistante :
⟹ *le firewall autorise des paquets ICMP de réponse à le franchir* ;

▷ la machine extérieure envoie du trafic ICMP d'erreur vers l'adresse publique de la machine intérieure
⟹ dans le cas d'une erreur TTL_EXPIRED, ce message d'erreur peut provenir de **n'importe quel routeur intermédiaire** ;
⟹ le firewall **laisse passer le trafic** en entrée et le **redirige vers la machine intérieure** ;
⟹ la machine intérieure **reçoit des données provenant de l'extérieur** !

Le «three-way handshake» :



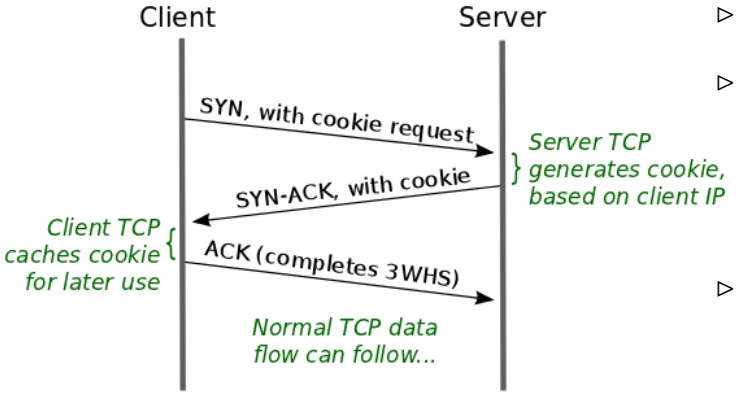
D'après la RFC 793 du protocole TCP :

- ▷ un segment SYN peut transporter des données ;
- ▷ **mais** elles ne peuvent être remises à l'application destinataire qu'**à la fin** du «handshake» ;
  pour des raisons de sécurité :
  - ◇ *un attaquant réalise une «spoofing attack» : il prend l'adresse IP d'une victime ;*
  - ◇ *il envoi un segment SYN avec données au serveur ;*
  - ◇ *le serveur répond avec des données avant la fin du «handshake» ;*
  - ◇ *la victime reçoit de(s) segment(s) de données de la part du serveur…*
- ▷ **mais** la programmation socket ne le permet pas.

http://lwn.net/Articles/508865/

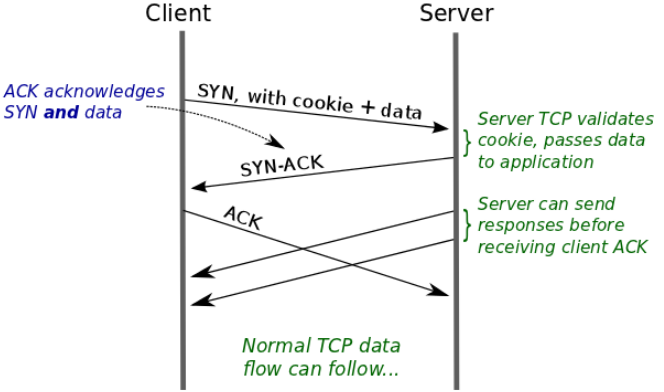
Avantages
- ⋆ échanger des ISN, «*Initial Sequence Number*» ;
- ⋆ échanger le MSS, «*maximum segment size*», la taille de la fenêtre, le «*window scale*», configuer le «SACK», obtenir des «*timestamps*» ;
- ⋆ éviter la création de connexions inutiles lors de l'envoi de segment SYN dupliqués ;

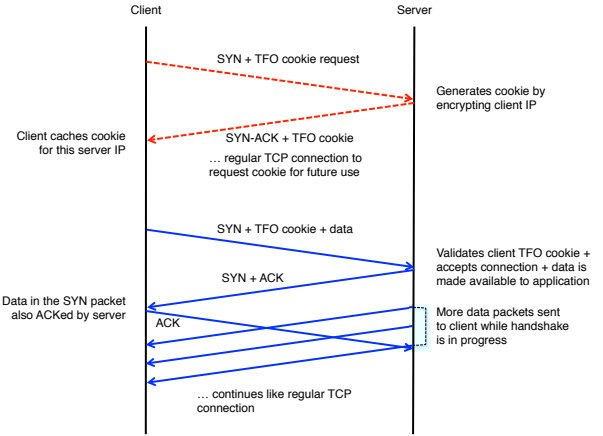
Inconvénients
- ◦ 1 RTT avant de débuter un échange pour l'établissement **d'une seule** connexion ;
- ◦ pour le protocole HTTP en version 1.1, les **connexions persistantes** :
  - ◇ mutualisent la connexion pour échanger différentes requêtes/réponses HTTP ;
  - ◇ **mais** sont rarement utilisées :
    - ⋆ le serveur, ou le firewall, coupe les communications inactives (le «*keep-alive*» de TCP est souvent insuffisant du fait de la cadence rapide des échanges) ;
    - ⋆ le navigateur Web établie de nombreuses connexions simultanées pour accélérer les échanges des différentes ressoures : CSS, scripts Javascripts, images *etc.*



- ▷ le client envoi un segment SYN + le cookie + des données ;
- ▷ le serveur **vérifie le cookie** : il chiffre l'adresse IP du client avec sa clé secrète :
  - ◇ si le cookie est **identique** : le serveur renvoi un SYN/ACK et des segments de données sans attendre de recevoir de ACK de la part du client ;
  - ◇ si le cookie est **différent** : le serveur détruit les données reçues et reprend le «three-way handshake» de TCP et renvoi un SYN/ACK ;
  - ◇ la connexion TCP se déroule **normalement** ;
- ▷ le serveur met à jour régulièrement la clé secrète pour rafraîchir les cookies : un «vieux» cookie ne peut servir à lancer une **attaque par rejeu**.

- ▷ le client demande le support de TFO et la création d'un TFO cookie sécurisé ;
- ▷ le serveur :
  - ◇ normalise l'adresse IP IPv4 ou IPv6 du client sur 16 octets ;
  - ◇ chiffre l'adresse IP obtenue avec un algorithme de chiffrement par block à l'aide d'une **clé secrète connu de lui seul** ⇒ MAC, «*Message Authentication Code*» ;
- ▷ le client reçoit et mémorise le cookie ;





- ▷ éliminer un RTT ;
- ▷ ne permet pas d'ignorer les segments SYN dupliqués : le serveur doit être **idempotent** :
  - ◇ **renvoyer le même résultat** pour **une même requête** ;
  - ◇ les serveurs Web fournissant un contenu statique ;
  - ◇ les serveurs Web fournissant un résultat dynamique suivant l'évolution un état interne **mais** capables d'analyser si deux requêtes sont identiques ;
- ▷ la **latence** diminue.

Au niveau de la programmation socket

Pour le serveur :

```
/* a hint value for the Kernel */
int qlen = 5;

/* create the socket */
fd = socket(AF_INET, SOCK_STREAM, 0);
/* bind the address */
bind(fd, addr, addrlen);

/* change the socket options to TCO_FASTOPEN */
setsockopt(sockfd, SOL_TCP, TCP_FASTOPEN, &qlen, si
zeof(qlen));

/* this socket will listen for incoming connections */
listen(fd, backlog);
```

pour le client :

```
/* create the socket */
fd = socket(AF_INET, SOCK_STREAM, 0);

/* connect and send out some data */
sendto(fd, buffer, buf_len, MSG_FASTOPEN, ...);

/* write more data */
send(fd, buf, size);

/* wait for some reply and read into a local buffer
*/
while ((bytes  = recv(fd, ...))) {
    ...
}
```


Au niveau du noyau avec `sysctl`

xterm

```
$ sudo echo 3 >/proc/sys/net/ipv4/tcp_fastopen
```

```
tcp_fastopen - INTEGER
  Enable TCP Fast Open feature (draft-ietf-tcpm-fastopen) to send data
  in the opening SYN packet. To use this feature, the client application
  must use sendmsg() or sendto() with MSG_FASTOPEN flag rather than
  connect() to perform a TCP handshake automatically.

  The values (bitmap) are
  1: Enables sending data in the opening SYN on the client w/ MSG_FASTOPEN.
  2: Enables TCP Fast Open on the server side, i.e., allowing data in
     a SYN packet to be accepted and passed to the application before
     3-way hand shake finishes.
  4: Send data in the opening SYN regardless of cookie availability and
     without a cookie option.
  0x100: Accept SYN data w/o validating the cookie.
  0x200: Accept data-in-SYN w/o any cookie option present.
  0x400/0x800: Enable Fast Open on all listeners regardless of the
     TCP_FASTOPEN socket option. The two different flags designate two
     different ways of setting max_qlen without the TCP_FASTOPEN socket
     option.
  Default: 1

  Note that the client & server side Fast Open flags (1 and 2
  respectively) must be also enabled before the rest of flags can take
  effect.

  See include/net/tcp.h and the code for more details.
```

https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

Connexion d'une machine Debian/Linux vers Google

```
20:10:39.920892 IP (tos 0x0, ttl 64, id 6594, offset 0, flags [DF], proto TCP (6), length 158)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [S], cksum 0x5c7d (incorrect -> 0x9bd5), seq 1779163941:1779164027,
win 29200, options [mss 1460,sackOK,TS val 114031335 ecr 0,nop,wscale 7,exp-tfo cookie 32a8b7612cb5ea57],
length 86

20:10:39.923005 IP (tos 0x0, ttl 57, id 3023, offset 0, flags [none], proto TCP (6), length 52)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [S.], cksum 0xae4c (correct), seq 1775907905, ack 1779164028,
win 42900, options [mss 1430,nop,nop,sackOK,nop,wscale 6], length 0

20:10:39.923034 IP (tos 0x0, ttl 64, id 6595, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [.], cksum 0x5c07 (incorrect -> 0x95af), ack 1, win 229,
length 0

20:10:39.923462 IP (tos 0x0, ttl 57, id 3024, offset 0, flags [none], proto TCP (6), length 589)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [P.], cksum 0xcdd1 (correct), seq 1:550, ack 1, win 670,
length 549

20:10:39.923475 IP (tos 0x0, ttl 57, id 3025, offset 0, flags [none], proto TCP (6), length 40)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [F.], cksum 0x91d0 (correct), seq 550, ack 1, win 670,
length 0

20:10:39.923492 IP (tos 0x0, ttl 64, id 6596, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [.], cksum 0x5c07 (incorrect -> 0x9382), ack 550, win 237,
length 0

20:10:39.923690 IP (tos 0x0, ttl 64, id 6597, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [R.], cksum 0x5c07 (incorrect -> 0x937d), seq 1, ack 551,
win 237, length
```

Explications :

- *l'adresse IP du client est* `106.186.29.14` *et celle du serveur* `173.194.38.98`*;*
- *l'option «*`exp-tfo`*» signifie «*experimental tcp fast open»
- *la longueur du premier paquet, 86 octets (une requête HTTP), alors qu'elle est normalement nulle, sans Fast Open ;*
- *le cookie est «*`32a8b7612cb5ea57`*» ;*
- *le serveur répond immédiatement «* `[P.]` *» avec la ressource d'une taille de 549 puis mets fin à la connexion «[F.]» ;*
- *le serveur n'acquitte que de 1 octet «*`ack 1`*», vu que les données du client étaient dans le paquet SYN.*
- *les traces sont fournies par Stéphane Bortzmeyer, <bortzmeyer@nic.fr>*

Le client demande l'activation de l'option et la récupération du cookie

```
16:53:26.120293 IP (tos 0x0, ttl 64, id 55402, offset 0, flags [DF], proto TCP (6), length 64)
    106.186.29.14.57657 > 74.125.226.86.80: Flags [S], cksum 0x07de (incorrect -> 0xc67b), seq
3854071484, win 29200, options [mss 1460,sackOK,TS val 325168854 ecr 0,nop,wscale 6,exp-tfo cookie
req], length 0

16:53:26.121734 IP (tos 0x0, ttl 57, id 16732, offset 0, flags [none], proto TCP (6), length 72)
    74.125.226.86.80 > 106.186.29.14.57657: Flags [S.], cksum 0xb913 (correct), seq 2213928284, ack
3854071485, win 42540, options [mss 1430,sackOK,TS val 2264123457 ecr 325168854,nop,wscale 7,exp-tfo
cookie 234720af40598470], length 0
```

Explications :

- ▷ *Le segment SYN du client est vide (pas de données) ;*
- ▷ *Le cookie est* `234720af40598470`*.*

Le client dispose du cookie et s'en sert pour une nouvelle requête

```
16:54:30.200055 IP (tos 0x0, ttl 64, id 351, offset 0, flags [DF], proto TCP (6), length 161)
    106.186.29.14.57659 > 74.125.226.86.80: Flags [S], cksum 0x083f (incorrect -> 0x651d), seq
1662839861:1662839950, win 29200, options [mss 1460,sackOK,TS val 325184874 ecr 0,nop,wscale 6,exp-
tfo cookie 234720af40598470], length 89

16:54:30.201529 IP (tos 0x0, ttl 57, id 52873, offset 0, flags [none], proto TCP (6), length 60)
    74.125.226.86.80 > 106.186.29.14.57659: Flags [S.], cksum 0x67e3 (correct), seq 2010131453, ack
1662839951, win 42540, options [mss 1430,sackOK,TS val 2264192396 ecr 325184874,nop,wscale 7], length
0
```

- ▷ *Le segment SYN du client contient les données de la requête ;*

Contraintes sur la génération du cookie par le serveur

- Lier le cookie à l'adresse IP source (pour éviter qu'un attaquant utilise le cookie d'un autre) ;
- Aller vite (le but est de diminuer la latence : calcul cryptographiques rapide, comme AES sur l'@IP client) ;
- Imposer une date d'expiration au cookie (soit en changer la clé privée utilisée lors de la génération, soit en incluant une estampille temporelle dans les données qui servent à générer le cookie) ;
- Vérifier un cookie entrant : refaire tourner l'algorithme et voir si on obtient le même résultat.

IPv4 est condamné : extinction des adresses IPv4 disponibles

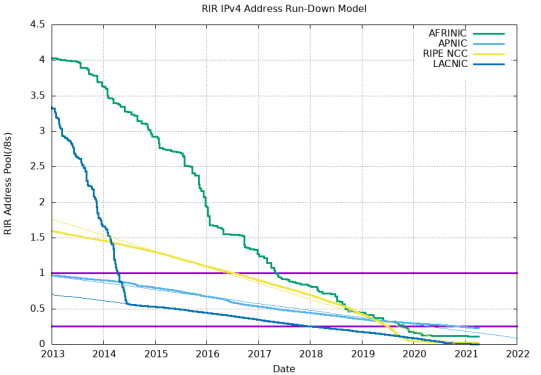
Internet est victime de son propre succès après plus de 30 ans d'existence (standardisé en 1981, RFC 791).

La croissance du nombre d'équipement connectés aux réseaux est exponentielle : ordinateurs, agents communicants, fusion annoncée de l'audiovisuelle et du réseau…

Les techniques :
- ⋆ CIDR, utiliser au mieux les adresses existantes ;
- ⋆ NAT, «*Network Address Translation*», cacher des réseaux entiers derrière une adresse unique ;

permettent de repousser la fin de IPv4, mais elle est **inéluctable** :

```
IANA Unallocated Address Pool Exhaustion: 03-Feb-2011
Projected RIR Address Pool Exhaustion Dates:
    RIR   Projected Exhaustion Date   Remaining Addresses
                                    in RIR Pool (/8s)
    APNIC:      19-Apr-2011 (actual)   0.2243
    RIPE NCC:   14-Sep-2012 (actual)   0.0201
    LACNIC:      10-Jun-2014 (actual)
    ARIN:      24 Sep-2015 (actual)   0.0003
    AFRINIC:   31-Dec--1              0.1108
```



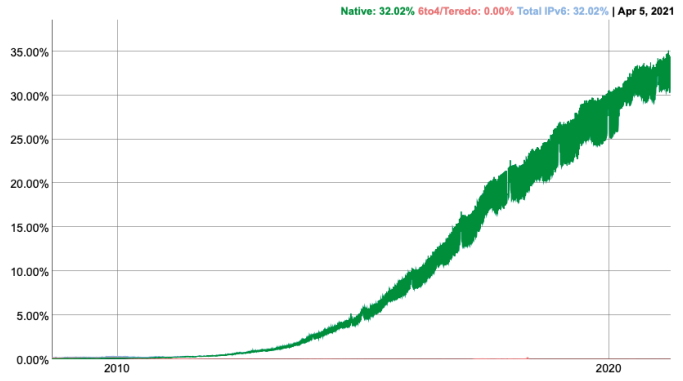
À l'adresse
`http://www.potaroo.net/tools/ipv4/index.html`,
on peut suivre «en direct» l'évolution des dernières adresses IPv4 disponibles.

Un problème d'adressage ?

Une adresse sur 32 bits donne 4 milliards d'adresses potentielles...mais il y a beacoup de gaspillage !

Les objectifs d'IPv4 ne sont plus adaptés :
- un adressage universel et suffisant : chaque interface réseau possède un numéro unique sur 32 bits ;
- méthode du «*Best Effort*» : le protocole fait de son mieux pour délivrer les paquets mais ne garantit rien pour les couches supérieures (4 et supérieure) que ce soient en :
  - ◇ pourcentage de paquets délivrés ;
  - ◇ en délai d'acheminement. *IPv4 n'intègre pas de QoS (Quality of Service).*

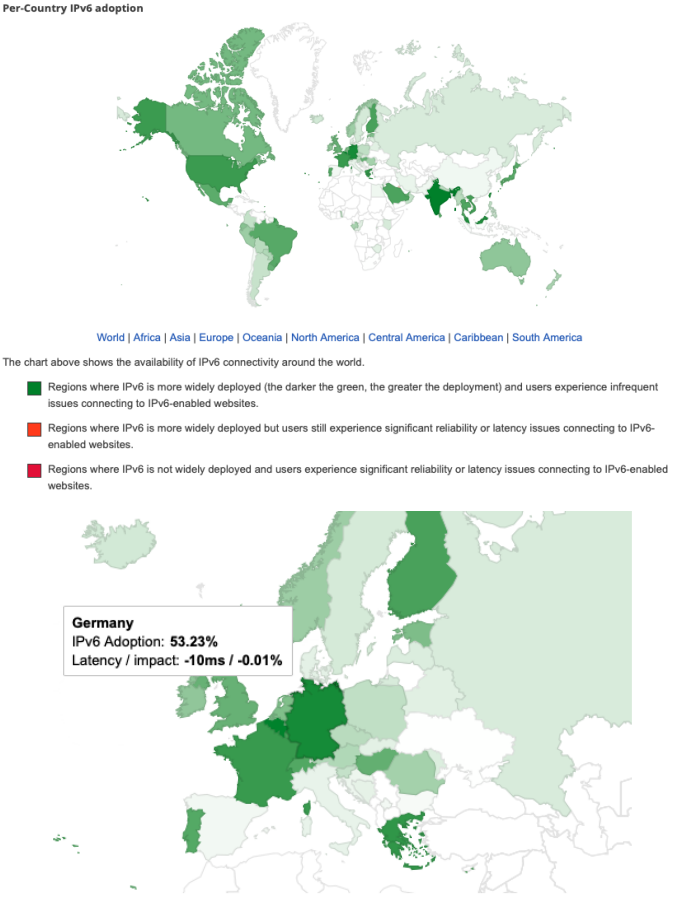
IPv4 le statut actuel en 2021



https://www.google.com/intl/en/ipv6/statistics.html

Ces statistiques sont enregistrées par Google suivant les accès réalisés par ses utilisateurs.

Il existerait en 2016, déjà plus de 20 milliards d'objets connectés, soit plus que les 4 milliards d'adresses IPv4 disponibles !



IPv4 est un protocole **sans connexion** (connectionless) :

- chaque paquet est transmis **indépendamment** des autres (il contient seulement l'adresse source et destination) ;
- aucun des paquets n'est **numéroté**, ni **marqué** comme appartenant à un flux ;
- si un paquet est perdu, il n'est pas possible de corriger l'erreur à ce niveau (couche 3) ;
- s'il est remis, il n'est **pas** possible de **prévoir le temps d'acheminement** nécessaire ;
- chaque nœud du réseau **fait de son mieux** pour délivrer le paquet dans un minimum de temps mais ne garantit pas lorsqu'il sera réellement délivré.

Des besoins et contextes nouveaux

- □ de nombreuses applications réclament une **QoS garantie** ;
- □ des réseaux ATM (mode connecté et rapide) où la **QoS peut être garantie** sont utilisés pour acheminer les paquets sans que la couche 3 en tire partie.

Des solutions restreintes :

- ▷ le protocole **RSVP**, «*Ressource Reservation Protocol*», permet d'allouer des ressources sur les routeurs afin de garantir une QoS ;
- ▷ le protocole **MPLS**, «*Multi Protocol Label Switching*», permet d'identifier les paquets par rapport à des flux et de les acheminer différemment en fonction de leurs besoins.

Pv6, la relève

IPv6 correspond à IPng «*Next Generation*».

Le travail de conception :

- ⋆ commencé en 1991 ;
- ⋆ achevé pour la plus grande partie en 1996 (publications des RFCs).

Cahier des charges pour le remplaçant de Ipv4

- **supporter des milliards** d'ordinateur, en se libérant de l'inefficacité de l'espace des adresses IP (classe de réseaux) ;
- **réduire la taille** des tables de routage (utilisation de mécanismes d'agrégation) ;
- **simplifier le protocole**, pour permettre aux routeurs de router les datagrammes plus rapidement ;
- fournir une **meilleure sécurité** (authentification et confidentialité) ;
- tenir compte du **type de service**, en particulier pour les services en temps réel ;
- faciliter la **diffusion** multi-destinataire ;
- donner la possibilité à un ordinateur de **se déplacer** sans changer d'adresse ;
- permettre des **évolutions futures** du protocole ;
- autoriser une **coexistence pacifique** avec l'ancien protocole (Ipv4).

Le choix du protocole SIPP «*Simple Internet Protocol Plus*» pour IPv6

Proposé par Francis et Deering en 1993.

Il est compatible avec les protocoles Internet de transport TCP et UDP, ce qui le rend potentiellement compatible avec les applications existantes.

IPv6 pas suffisamment convaincant

- Pas de «*Killer application*» :
  - la mobilité ? IPv4 l'intègre… (VPN, IPMobilev4, *etc.*)
  - la sécurité ? IPv4 l'intègre… (IPSec, *etc.*)
  - l'auto-configuration ? IPv4 l'intègre… (DHCP, ZeroConf, PIPA, *etc.*)
- Le coût du passage à IPv6 : changer les matériels, mettre à jour *etc.*
- IPv4 est «*scalable*» : utilisation du CIDR, du NAT pour toujours plus de croissance ;
- Plus d'adresses possibles dans IPv6 ? Mais le jour de la connexion de tous les appareils électriques n'est pas encore arrivé !

Les avantages décisifs d'IPv6

- Beaucoup plus d'adresses ;
- Une adresse plus grande : meilleure intégration de la répartition géographique et par catégorie d'usage pour le «subnetting» ;
- Routage simplifié et «*scalable*» ;
- Respect du principe «*end-to-end*» ;

$\Longrightarrow$ **IPv6 reste l'avenir d'IPv4**

Combien d'adresses nécessaires ?

Evaluation des besoins :

- 10 Milliards de personne d'ici 2050 ;
- chaque personne a plus d'un ordinateur ou d'un matériel connecté au réseau :
  Si 100 ordinateurs par personne alors 10^{12} ordinateurs (IoT, téléphone mobile *etc.*)
- mais plus d'adresses sont nécessaires :
  - plusieurs interfaces par noeuds ;
  - plusieurs adresses par interface ;
  - 2^6 ou 2^8 adresses par noeud…

  marge de sécurité : 10^{15} adresses Pré-requis : 10^{12} machines et 10^9 réseaux, et pourquoi pas 10^{12} à 10^{15} réseaux dans le futur !

Le choix

des adresses sur 128 bits soit 4 fois la taille d'une adresse Ipv4 (32 bits) :

- $2^{128} = 3,410^{38}$ adresses ;
- $665 * 10^{21}$ adresses par m^2 à la surface de la Terre !

Ce qui permet :

- plusieurs interfaces par hôtes ;
- **plusieurs adresses** par interface ;
- des adresses **Unicast** (un vers un), **Multicast** (un vers plusieurs), **Anycast** (un vers un choisi parmi un groupe : *le plus proche*) ;
- des adresses par «**fournisseur d'accès**», propres à un **site** (adresses privées), propres à un **lien** de communication (link-local) ;
- une partie de l'adresse IPv6 peut être l'**adresse MAC** (IEEE802) ⟶ auto-configuration !

Une nouvelle notation

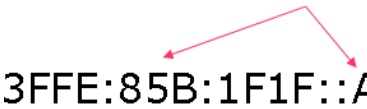
3FFE:085B:1F1F:0000:0000:0000:00A9:1234

Une adresse décomposée en **8 groupes** de 16 bits

- ▷ séparés par des «:» ;
- ▷ notés en **hexadécimal**.

Les zéros en début de groupe peuvent être omis.

Les séquences de zéro, c-à-d plusieurs groupes à zéro, peuvent être omises (mais une seule fois, pas de «::» deux fois !)



3FFE:85B:1F1F::A9:1234

Pour la compatibilité avec IPv4 :

⇒ on peut laisser les derniers 32bits en notation décimal séparée par des points (dot-decimal) : `::192.50.185.25`

Pour l'adresse de « loopback » :

⇒ 0:0:0:0:0:0:0:1 ⟶ ::1 ou ::1/128 (128 est le préfixe comme en IPv4 CIDR ou VLSM).

Dans une URL, on l'encadre entre crochets (RFC2732)

⇒ `http://[2001:1:4F3A::206:AE14]:8080/index.html`

Le datagramme IPv6 : une en-tête simplifiée

Les différences

Enlevé :

- ▷ **Identification**, **flags**, **flag offset** : plus de fragmentation ;
- ▷ **TOS** : plus de type de service ;
- ▷ **hlen** : taille d'en-tête fixe ;
- ▷ **header Checksum** : plus de détection d'erreur dans l'en-tête, plus rapide à traiter et les erreurs sont rares ;
- ▷ plus de champs optionnels.

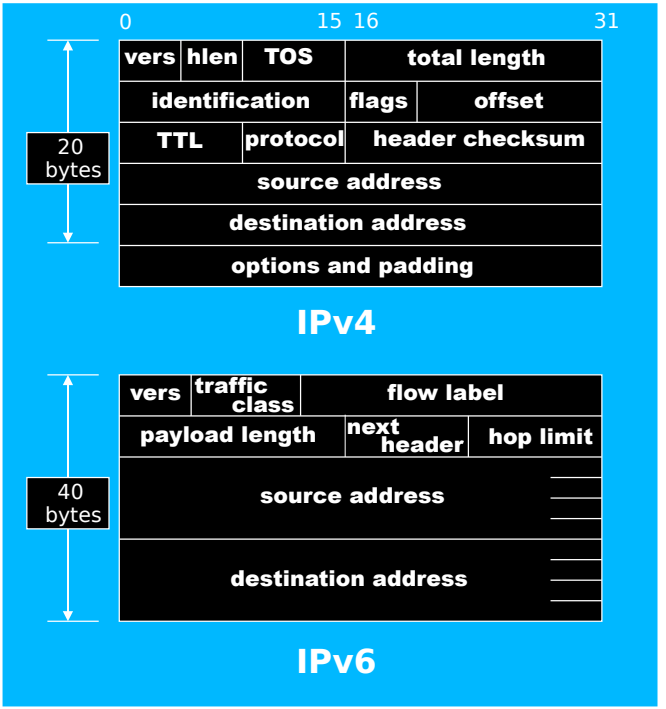
Changé :

- ▷ total length ⟶ **payload**, la taille des données n'est plus celle du datagramme moins celle de l'en-tête (plusieurs en-tête ?)
- ▷ protocol ⟶ **next header**, entête multiples !
- ▷ TTL ⟶ **hop limit**, ce qui est vrai…

Ajouté :

- ▷ **trafic class**, de la QoS ?
- ▷ **flow label**, associé des communications entre elles ?

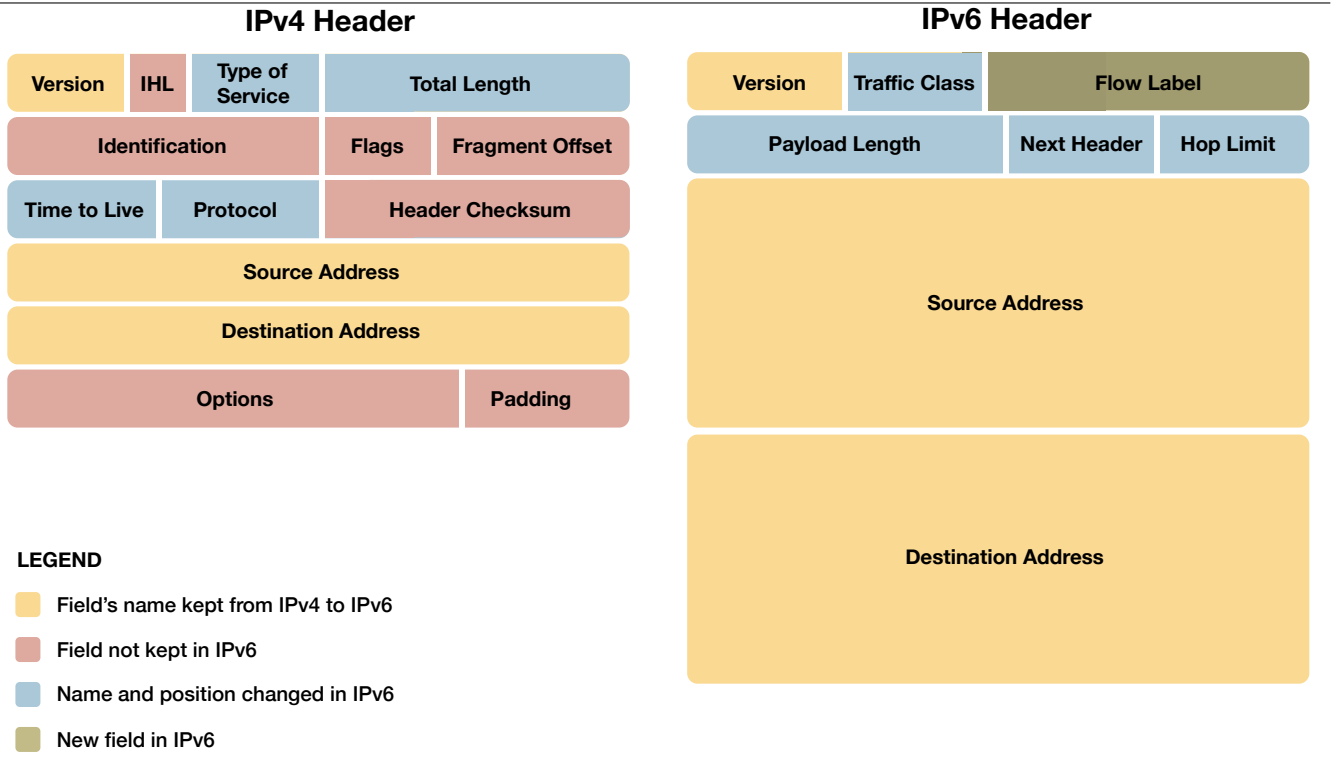
Étendu : les adresses !



Le but

- ▷ traitement **plus rapide** par les routeurs (format fixe et simple)
- ▷ **sans sacrifier** la flexibilité (ajout d'en-tête successives).

Différences entre datagramme IPv4 et IPv6



- le «*trafic class*» : sur 8bits permet de définir des priorités ou des classes de trafic, «*Differentiated Services*».
- le «*payload length*» : sur 16bits définit la taille des données, entête non comprise.

Taille du datagramme

La MTU, «*Maximum Transfert Unit*», choisie comme minimum pour acheminer un datagramme est indiquée dans la RFC du protocole :

- IPv4, RFC 791 : 576 octets avec une entête variant de 20 à 60 octets ;
  dans la pratique : 1500 octets avec des technologies Ethernet.
- IPv6, RFC 2460 : 1280 octets avec une entête fixe de 40 octets.

Les extensions deviennent des « entêtes d'extension »

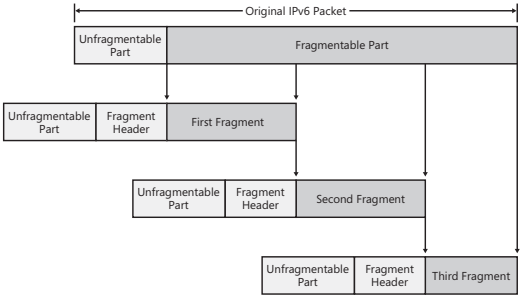
Le champ «*Next header*» indique le **type d'entête** suivant immédiatement l'entête Ipv6.

Ces extensions d'entête sont optionnelles :

- elles ne sont traitées par **aucun intermédiaire** lors de l'acheminement du datagramme, mais seulement par le destinataire (quelques exceptions, comme la «*hop by hop*» extension) ;
- elles sont identifiées par le champ «*next header*» de l'entête précédente ;
- elles sont de taille multiple de 8 octets et cette taille, « *hdr length*» est indiquée à la suite du champ « *next header*». *On utilise les mêmes valeurs que dans le champ «Protocol» de IPv4 pour référencer les protocoles de niveau 4 (TCP=6 , UDP=17)*
- la taille du datagramme IPv6 = somme de la taille de toutes les en-têtes et du contenu du paquet.

Différents types d'entête

- **Hop-by-Hop Header** : information examinée sur chaque noeud intermédiaire du chemin suivi par le datagramme IP ;
- **Destination Options Header** : options à examiner que par le destinataire du datagramme ;
- **Routing Header** : routage choisi par la source (strict ou faible, *loose or tight*) :
  - contient une liste un ou plusieurs noeuds intermédiaires "à visiter" au cours de l'acheminement du datagramme
  - chaque nœud intermédiaire modifie la liste pour passer au nœud suivant.
- **Fragment header** :
  envoi de paquets plus long que le MTU
  Attention : *dans IPv6, la fragmentation n'est réalisée que par la source.*
  La partie «unfragmentable» : les en-têtes IPv6, «hop by hop» et «Destination Options».
  La partie «fragmentable» : les en-têtes traitées que sur la destination : «Authentication», «ESP», celle du protocole de haut niveau (TCP ou UDP).



- **Authentication Header** : authentification et intégrité des données ;
- **Privacy Header** : chiffrement du contenu à protéger, ESP, «Encapsulating Security Payload».

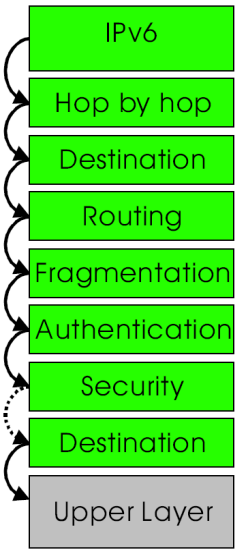
Exemple de combinaisons d'entêtes

Un segment TCP envoyé :

- ☐ seul ⟹

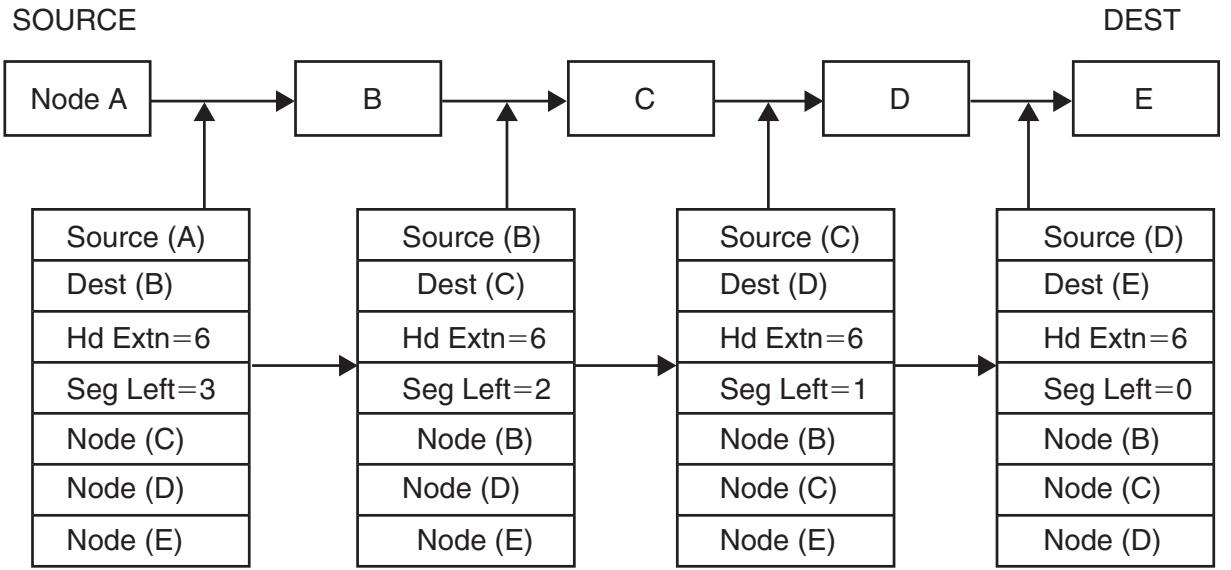
- ☐ avec un routage choisi par la source ⟹

- ☐ avec un routage choisi par la source et une fragmentation ⟹



- ▷ «*hop by hop*» : paramètres traités par chaque routeur ;
- ▷ «*Destination*» : traité par les routeurs de la liste de l'ext. «Routing»
- ▷ «*Routing*» : liste des routeurs à traverser ;
- ▷ «*Fragmentation*» : traité par la dest., après ré-assemblage des fragments ;
- ▷ «*Authentication*» : traité par la destination (signature électronique) ;
- ▷ «*Security*» : chiffrement du contenu du datagramme ;
- ▷ «*Destination*» : paramètres traités uniquement par la destination.

Exemple du «Rounting Header»



- ▷ le champ «Hd Extn» indique en mots de 64 bits la taille des adresses présentes *Il faut le diviser par 2 pour connaître le nombre d'adresses sur 128bits (ici, 6/2 = 3)*
- ▷ le champ «Seg Left» indique le nombre d'adresses restant à traiter ;
- ▷ à chaque passage par un intermédiaire, on échange l'adresse destination courante avec l'adresse à traiter, puis on décrémente de un le champ «Seg Left».

La fragmentation en IPv6

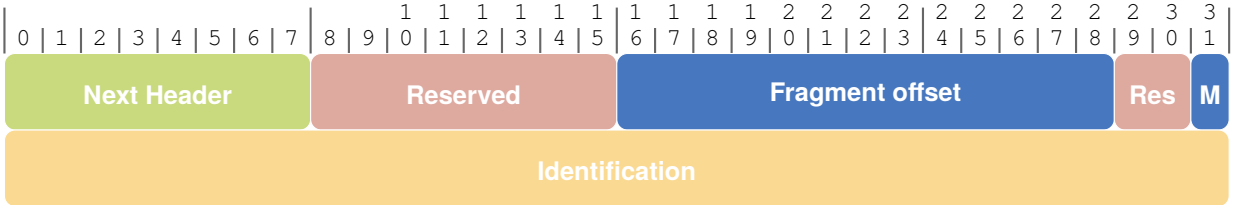
Par rapport à IPv4, seule la source peut fragmenter, les **routeurs ne le font jamais**.

⟹La source doit connaître la MTU du réseau que doit emprunter le datagramme :

- ▷ utilisation du «*path MTU discovery*», protocole de découverte de la MTU employée :
  - ◇ envoi d'un paquet ICMPv6 de la taille la plus grande autorisée par son lien ;
  - ◇ récupération d'un message ICMPv6 «packet size too big» ;
  - ◇ conservation de la valeur de la MTU pendant un certain temps avant d'être recalculée (environ 10mins).

Fonctionnement

- ▷ utilisation d'une entête ;
- ▷ le «*payload*» ou contenu du datagramme est décomposé en morceaux ;
- ▷ un nouvel entête est défini pour chaque morceau :



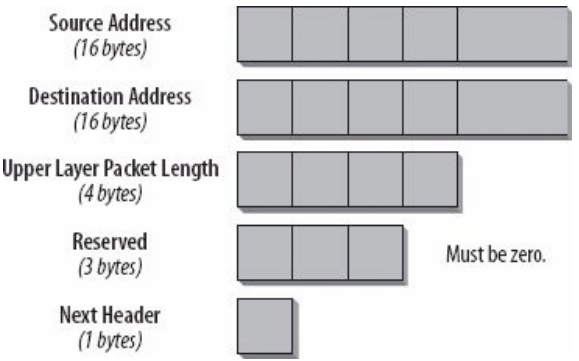
Par rapport à IPv4 :

- ◦ il n'y a pas de bit «*Don't fragment*» ;
- ◦ le champs **identification** est sur **32 bits** au lieu de 16bits ;
- ◦ les drapeaux de fragmentation, 3 bits, dont 1 seul utile MF util, sont situées en fin de «*flag offset*» au lieu du début ;
- ◦ le «*fragment offset*» reste sur 13bits.

Calcul du «cheksum» des protocoles TCP et UDP

- UDP et TCP sont des protocoles de niveau 4 (couche transport) ;
- ils utilisent des TSAP, «Transport Service Access Point» pour distinguer les différentes communications (multiplexage) composés chacun de :
  - une adresse IP (celle de l'interface utilisée et celle de l'interlocuteur) ;
  - un numéro de port (source et destination) ;
- pour savoir si des erreurs de transmission ont modifié les données, il faut **vérifier l'intégrité** du datagramme UDP ou du segment TCP :
  - le TSAP fait partie des données à vérifier (une erreur présente : les données sont ignorées) ;
  - l'adresse IP est une information de la couche de niveau 3 (IP) :
    - il faut «intégrer» l'@IP dans le calcul du checksum : utilisation d'une «pseudo-entête», «pseudo header» ;
    - en IPv6, il y a un nombre variable d'extensions : ne pas en tenir compte ;

Pseudo en-tête et calcul du checksum



Le

- @IPv6 source et destination sur 16 octets ;
- «*upper layer packet length*» : la taille du segment TCP ou du datagramme UDP ;
- «*next header*» : indique au choix UDP ou TCP avec la valeur 17 ou 6.

checksum, sur 16 bits, est calculé sur la pseudo-entête et le contenu du segment/datagramme.

***Remarque** : en IPv6, le calcul d'un checksum pour UDP est obligatoire contrairement à IPv4.*

Le champ «Trafic Class »»

utilisé comme en IPv4 (ou comme il aurait pu).

Le champ «Flow Label»

- ▷ permet de classifier des datagrammes appartenant à un flux spécifique ;
  - ◇ un flux est une **séquence de datagramme** qui doivent être **traités de manière spécifique** ;
  - ◇ ce flux est définie pour le même protocole de la couche «transport», TCP ou UDP : (TSAP source, TSAP destination, protocole)

Pour faire de la QoS : l'**analyse**, lors du routage, ne porte que sur l'étiquette de flot.

Sans ce champ, il aurait fallu réaliser le décodage du champ «*next header*» et l'analyse des numéros de port :

- □ gestion moins efficace des datagrammes ;
- □ impossible parfois si les données sont **chiffrées** (impossible d'analyser le contenu du datagramme).

Remarque : La présence du champ «*Flow Label*» n'est pas toujours utilisé au profit d'une mémorisation/étiquetage des datagrammes dans un firewall (QoS similaire à IPv4).

Découpage <partie réseau>.<partie machine>

Comme pour IPv4, une adresse IPv6 peut être découpée en une partie réseau et une partie machine par l'usage d'un **préfixe**.
Plusieurs types d'adresses ont été définies :

- ▷ en divisant les adresses en 2 : les 64 premiers bits désignant la partie réseau, les 64 derniers la machine hôte ;
- ▷ en utilisant un préfixe de valeurs fixées de bits pour la partie réseau pour indiquer le type d'adresse, RFC 3513.

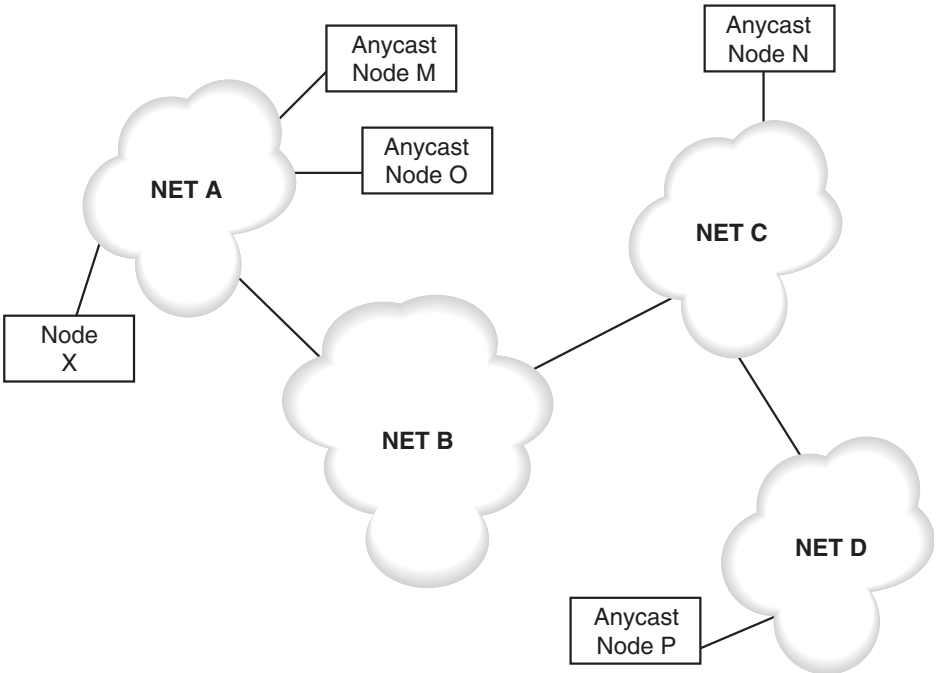
Différents types d'adresses en fonction des besoins

IPv6 offre la possibilité d'**attribuer plus d'une adresse** à une interface, chacune de ces adresses ayant un but bien précis, suivant une **portée** ou « scope » (lien, site, global) et une **cardinalité** (unicast, multicast).

- □ **unicast** : une adresse de ce type désigne une interface unique.
  Un datagramme envoyé à une telle adresse, sera donc remis à l'interface ainsi identifiée.
  Parmi les adresses unicast, on peut distinguer celles :
  - ◇ qui auront une **portée globale**, c-à-d désignant sans ambiguïté une machine sur Internet ;
  - ◇ celles qui auront une **portée locale** (lien ou site).
    Ces dernières ne pourront pas être routées sur l'Internet.

- □ **multicast** : une adresse de type multicast désigne un **groupe d'interfaces** qui en général appartiennent à des noeuds différents pouvant être situés n'importe où dans Internet.
  Lorsqu'un datagramme a pour destination une adresse de type multicast, il est acheminé par le réseau à toutes les interfaces membres de ce groupe.
  Il n'y a plus d'adresses de type broadcast comme sousIPv4 ; elles sont remplacées par des adresses de type multicast qui saturent moins un réseau local constitué de commutateurs (switch).

- **anycast** : choix du destinataire le plus proche parmi un groupe défini par plusieurs interfaces disposant de la même adresse unicast qui ne peut être utilisée qu'en destination d'un datagramme.
  Exemple l'adresse anycast d'un routeur : `<préfixe réseau sur 64 bits>::/128`.



Ici, le nœud X transmet un datagramme en Anycast et reçoit une réponse de M et O, plus proches que N et P.

En IPv4, des préfixes existent pour définir les différentes classes réseaux :

- ▷ 0xxxxx...xxx permet de définir un réseau de classe A ;
- ▷ 10xxx...xxxx permet de définir un réseau de classe B ;
- ▷ 110xxx...xxx permet de définir un réseau de classe C ;
- ▷ 1110xx...xxx permet de définir un réseau de classe D.

En IPv6, on définit un certain nombre de préfixes qui permettent de définir les types d'adresses suivants (RFC 4291) :

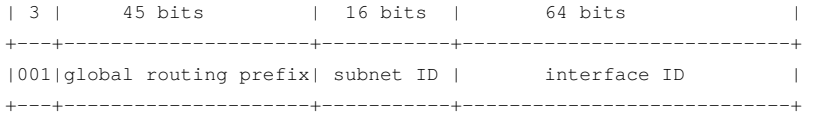
Address type	Binary prefix	IPv6 notation
Unspecified	00...0 (128 bits)	::/128
Loopback	00...1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-Local unicast	1111111010	FE80::/10
Global Unicast	(everything else)	

***Attention** : les adresses* `Site-local unicast 1111111011 FEC0::/10`*, RFC 3513, sont obsolètes et remplacées par les «Unique Local IPv6 Unicast Addresses» définies dans la RFC 4193, de préfixe* `FC00::/7` *que l'on verra plus loin.*

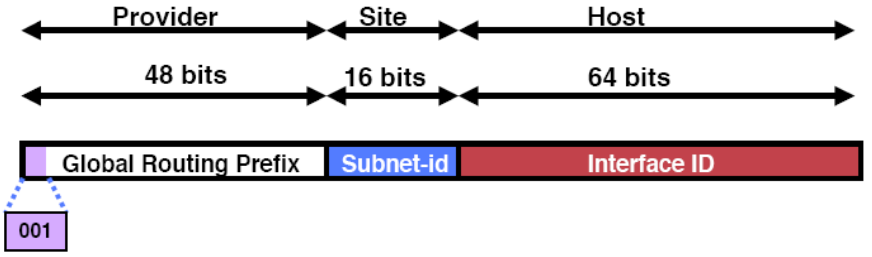
Pour les adresses «*Global Unicast*» :

Tout ce qui est **différent** de `000::/3` est une adresse unicast globale, RFC3587

Actuellement, les adresses globales unicast sont définies comme suit :



Pour tout savoir sur l'actualité des adresses IPv6 : **http://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.xhtml**



Ce schéma est noté 2000 : :/3.

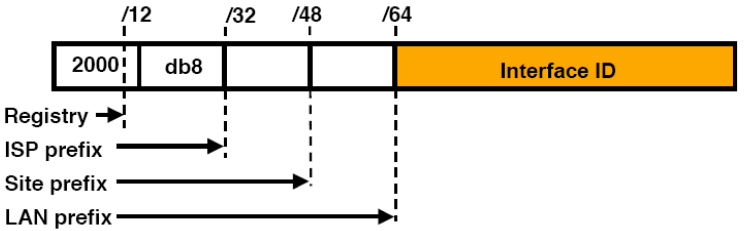
Ces adresses permettent le routage à travers Internet :

- elles sont globales, connues de tous ;
- elles permettent une organisation hiérarchique (agrégation en réseau, sous-réseau, machine).

Un nouveau schéma d'**agrégation** pour tirer parti de la grandeur des adresses :

- les accès Internet se répartissent en énormes FAI, « fournisseurs d'accès intenet » ou ISP ;
- ces FAI fournissent à de plus petits FAI ;
- et ainsi de suite jusqu'au particulier.

Cette structure facilite la migration d'un FAI à un autre : il suffit de changer le préfixe !



L' IANA alloue les adresses 2000::/3 de la manière suivante :

- chaque domaine d'enregistrement obtient un préfixe unique sur 12 bits de l'IANA ;
- ensuite chaque domaine d'enregistrement alloue un préfix sur 32 bits ou plus pour un FAI (ISP Internet Service Provider) ;
- chaque FAI peut alors fournir un préfix sur 48 bits pour chaque client ;

Ainsi :

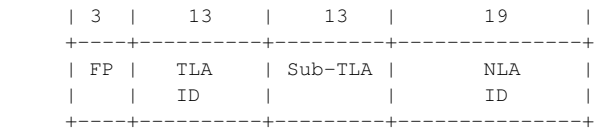
- ▷ chaque client dispose de 64 bits pour identifier chacune de ces machines (interface ID basée sur l'@MAC) ;
- ▷ 16 bits sont réservés pour la création de sous réseau chez le client (Site prefix) ;
- ▷ 16 bits sont réservés pour le FAI pour identifier chaque client (ISP Prefix) ;
- ▷ 20 bits sont réservés pour définir des FAI différents !

Les adresses IPv6 sont allouées actuellement par 5 RIRs:

- ◇ AfriNIC,
- ◇ APNIC,
- ◇ ARIN,
- ◇ LACNIC,
- ◇ RIPE NCC

Ensuite :

- ▷ les FAIs obtiennent leur espace d'adresses des RIRs ;
- ▷ les entreprises obtiennent leur espace d'adresses IPv6 de leur FAI.



where:

FP = 001 = Format Prefix This is the Format Prefix used to identify aggregatable global unicast addresses.

TLA ID = 0x0001 = Top-Level Aggregation Identifier This is the TLA ID assigned by the IANA for Sub-TLA allocation.

Sub-TLA ID = Sub-TLA Aggregation Identifier The IANA will assign small blocks of Sub-TLA ID's to registries. The registries will assign the Sub-TLA ID's to organizations.

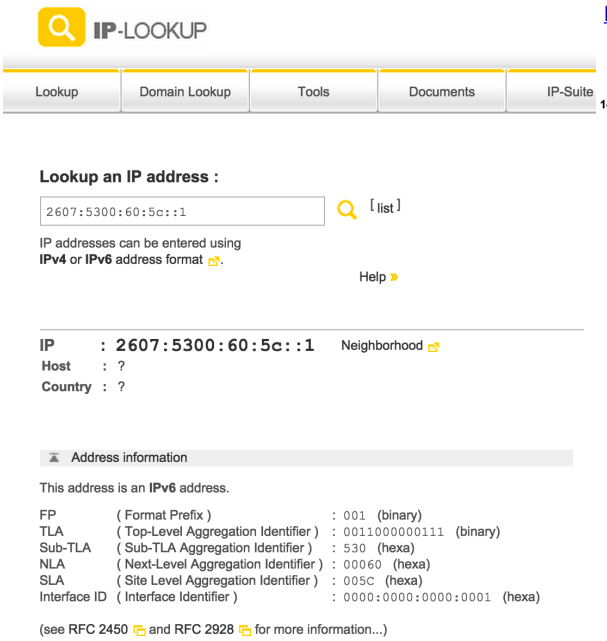
NLA ID = Next-Level Aggregation Identifier

Next-Level Aggregation ID's are used by organizations assigned a TLA ID to create an addressing hierarchy and to identify sites.

Binary Value	IPv6 Prefix Range	Assignment
0000 000X XXXX X	2001:0000::/29 - 2001:01F8::/29	IANA
0000 001X XXXX X	2001:0200::/29 - 2001:03F8::/29	APNIC
0000 010X XXXX X	2001:0400::/29 - 2001:05F8::/29	ARIN
0000 011X XXXX X	2001:0600::/29 - 2001:07F8::/29	RIPE NCC
0000 100X XXXX X	2001:0800::/29 - 2001:09F8::/29	(future assignment)
0000 101X XXXX X	2001:0A00::/29 - 2001:0BF8::/29	(future assignment)
. . .		
. . .		
. . .		
1111 111X XXXX X	2001:FE00::/29 - 2001:FFF8::/29	(future assignment)

Where "X" indicates "0" or "1".

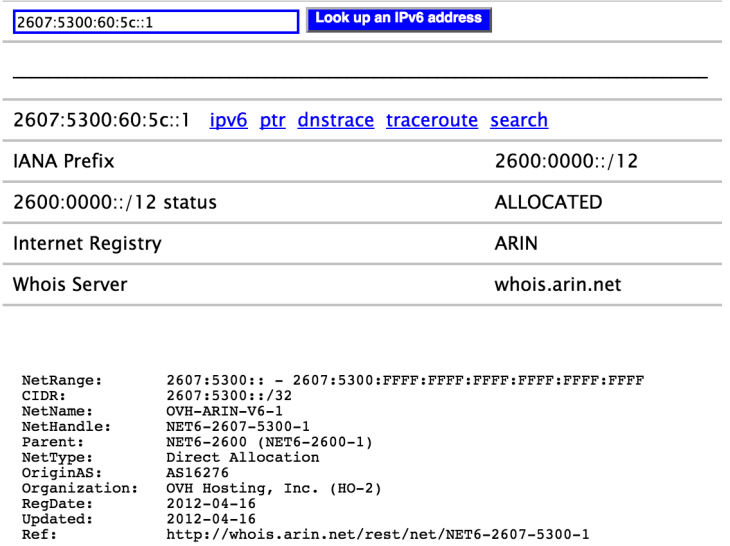
Le site «libpfb.so»



http://ip-lookup.net/index.php

IPduh.com / ipv6 / whois / ? 2607:5300:60:5c::1

1423420268>



http://ipduh.com/ipv6/whois/

Affectation des préfixes actuels

Prenons l'exemple de OVH :

HURRICANE ELECTRIC INTERNET SERVICES

Search

AS16276 OVH SAS

Quick Links
- BGP Toolkit Home
- BGP Prefix Report
- BGP Peer Report
- Bogon Routes
- World Report
- Multi Origin Routes
- DNS Report
- Top Host Report
- Internet Statistics
- Looking Glass

AS Info | Graph v4 | Graph v6 | Prefixes v4 | Prefixes v6 | Peers v4 | Peers v6 | Whois | IRR

Prefix	Description
2001:41d0::/32	OVH SAS
2402:1f00::/32	OVH
2607:5300::/32	OVH Hosting, Inc.

Updated 07 Feb 2015 07:28 PST © 2015 Hurricane Electric

IPduh.com / ipv6 / whois / ? 2402:1f00::1

1423420647> 2402:1f00::1 Look up an IPv6 address

2402:1f00::1 ipv6 ptr dnstrace traceroute search

IANA Prefix	2400:0000::/12
2400:0000::/12 status	ALLOCATED
Internet Registry	APNIC
Whois Server	whois.apnic.net

```
inet6num:       2402:1F00::/32
netname:        OVH-AP
descr:          OVH
descr:          2 Rue Kellermann
country:        HK
admin-c:        ON17-AP
tech-c:         ON17-AP
notify:         noc@ovh.net
mnt-by:         APNIC-HM
mnt-lower:      MAINT-AP-OVH
mnt-routes:     MAINT-AP-OVH
mnt-irt:        IRT-OVH-AP
status:         ALLOCATED PORTABLE
remarks:        -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
remarks:        This object can only be updated by APNIC hostmasters.
remarks:        To update this object, please contact APNIC
remarks:        hostmasters and include your organisation's account
remarks:        name in the subject line.
remarks:        -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
changed:        hm-changed@apnic.net 20111004
source:         APNIC
```

		01-12			
2400:0000::/12	APNIC	2006-10-03	whois.apnic.net	ALLOCATED	2400:0000::/19 was allocated on 2005-05-20. 2400:2000::/19 was allocated on 2005-07-08. 2400:4000::/21 was allocated on 2005-08-08. 2404:0000::/23 was allocated on 2006-01-19. The more recent allocation (2006-10-03) incorporates all these previous allocations.
2600:0000::/12	ARIN	2006-10-03	whois.arin.net	ALLOCATED	2600:0000::/22, 2604:0000::/22, 2608:0000::/22 and 260c:0000::/22 were allocated on 2005-04-19. The more recent allocation (2006-10-03) incorporates all these previous allocations.
2610:0000::/23	ARIN	2005-11-17	whois.arin.net	ALLOCATED	
2620:0000::/23	ARIN	2006-09-12	whois.arin.net	ALLOCATED	
2800:0000::/12	LACNIC	2006-	whois.lacnic.net	ALLOCATED	2800:0000::/23 was allocated on 2005-11-17. The more recent

http://www.iana.org/assignments/ipv6-unicast-address-assignments/ipv6-unicast-address-assignments.xhtml

La région du réseau correspond au RIR, «Regional Internet Registry»*, associé :* `2400` *pour Honk-Kong et l'APNIC,* `2600` *pour le RIPE.*

Diminution de la taille des tables de routage

Pour définir les tables de routage :

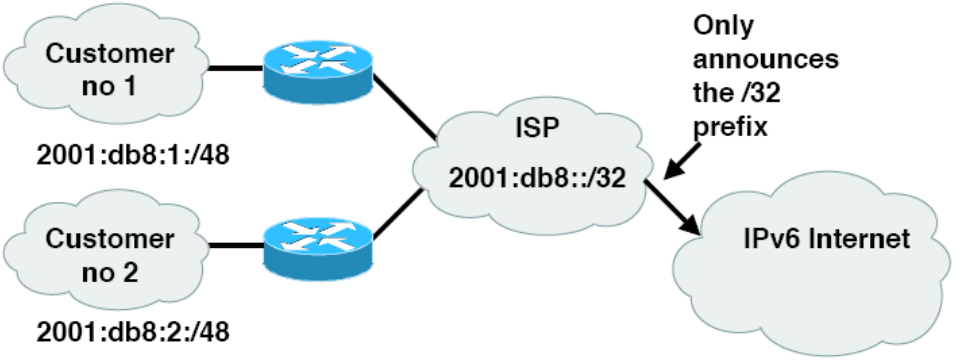
▷ on bénéficie du mécanisme de «l'agrégation» ;

Exemple :

⋆ le routage se fait suivant le préfixe /32 vers le FAI (ISP) ;

⋆ le routage se fait suivant le préfixe /48 à l'intérieur du FAI ;

⋆ regroupement des clients (Customer 1 & 2) auprès de leur FAI (ISP).



Utilisation de la commande «dig» avec le paramètre «AAAA» :

xterm

```
pef@darkstar:~ dig aaaa libpfb.so

; <<>> DiG 9.8.3-P1 <<>> aaaa libpfb.so
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36803
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;libpfb.so.       IN AAAA

;; ANSWER SECTION:
libpfb.so.     13502 IN AAAA  2607:5300:60:5c::1

;; Query time: 1237 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Feb  8 19:28:35 2015
;; MSG SIZE  rcvd: 55
```

Suivant les systèmes d'exploitation et la connexion Internet dont on dispose (proposant IPv6 ou non), ce sera l'adresse IPv6 de destination qui sera privilégiée par rapport à celle IPv4.

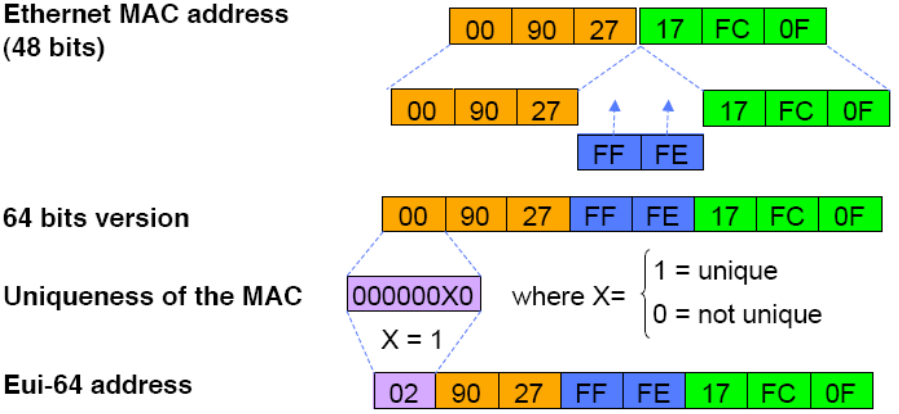
L'identifiant d'interface ou «interface identifier»

Il est sur 64 bits pour être compatible avec le IEEE 1394 (FireWire).

Il est nommé EUI-64.

Il peut être défini à partir de l'adresse MAC Ethernet sur 48 bits :

- cette adresse est décomposée en : <id. constructeur sur 3 octets>.<id. interface sur 3 octets> ;
- elle est **unique ou non** : 7ème bit de l'@MAC à 1 si elle est unique ;
- on la complète en **ajoutant deux octets** pour arriver à 6 octets en tout et en **inversant** le 7ème bit :



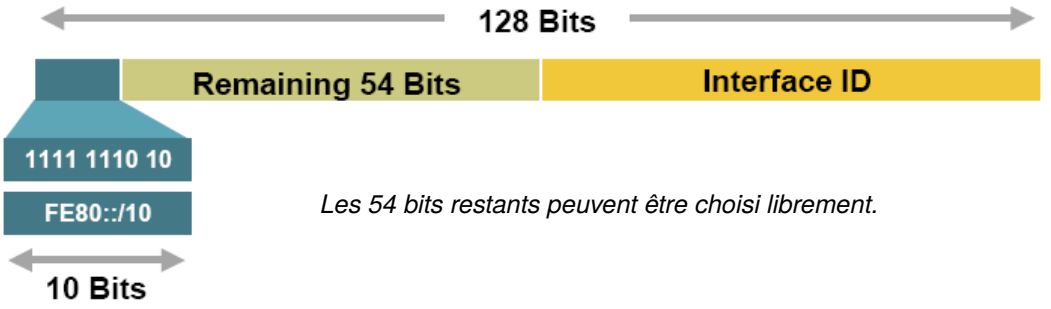
L'**interface identifier** peut être également choisi :

- de manière aléatoire : «Security identifier», (éviter le traçage des utilisateurs par exemple) RFC 3041, «*Privacy Extensions for Stateless Address Autoconfiguration in IPv6*» ;
- défini par DHCPv6 ;
- configuré manuellement.

Adresse unicast de lien ou «*link-local*»

Ce type d'adresse n'est **valide** que sur le **lien** d'une interface :

▷ un datagramme avec une **adresse destination** de ce type **ne franchira jamais** un routeur.



Les 54 bits restants peuvent être choisi librement.

Notation «*zone index*» : <adresse ipv6> **%** <interface>

`fe80::7479:97ff:fea5:995d%eno1` ⟹ utiliser cette adresse sur l'interface `eno1`

Envoyer un ping vers cette machine accessible par `eno1` :

```
xterm
$ ping6 fe80::7479:97ff:fea5:995d%eno1
```

▷ **découvrir l'existence** d'autres machines sur un même lien, avec le protocole de **découverte de voisins**, «*neighbor discovery*» qui remplace ARP ⟹ **ARP n'existe plus en IPv6** ;

▷ **configurées automatiquement** à l'initialisation de l'interface en la dérivant depuis l'adresse MAC ou par tirage aléatoire ;

▷ faciliter la **configuration d'adresse globale** : le préfixe `FE80` de 64bits est **remplacé** par un préfixe réseau global.

Une adresse de type **multicast** désigne un **groupe d'interfaces** qui appartiennent, en général, à des noeuds différents répartis dans le réseau.
Lorsqu'un datagramme a pour destination une adresse de type multicast, il est acheminé par le réseau à toutes les interfaces membres de ce groupe.

Les adresses multicast **remplacent** également le **broadcast** d'IPv4 $\Longrightarrow$ **plus de broadcast en IPv6 !**

Lifetime	
0	If Permanent
1	If Temporary

Scope	
1	Node
2	Link
5	Site
8	Organization
E	Global

- `ffx1` : **noeud-local**, ces paquets ne quittent jamais le noeud (utile pour des applications locales au nœud ;
- `ffx2` : **lien-local**, ces paquets ne sont jamais transmis par les routeurs, ils ne quittent par conséquent jamais le lien spécifié ;
- `ffx5` : **site-local**, ces paquets ne quittent jamais le site ; $\Longleftarrow$ **abandonné !**
- `ffx8` : **organisation-locale**, ces paquets ne quittent jamais l'organisation ; $\Longleftarrow$ **abandonné !**
- `ffxe` : **portée globale**, toutes les machines qui peuvent être atteintes ;
- *les autres sont réservés.*

Utiliser des adresses **multicast** spécifiques pour joindre certaines machines d'un réseau suivant **les services** qu'ils proposent et leur **nature**.

On utilise le champs «group ID» sur 112bits où l'on fixe les 80 premier bits à zéro, ce qui laissent 32 bits à choisir.

Exemples

Les noeuds Ipv6 :

- `FF01:0:0:0:0:0:0:1` all IPv6 nodes, within scope 1 (interface-local) ⟹ même machine
- `FF02:0:0:0:0:0:0:1` all IPv6 nodes, within scope 2 (link-local)⟹toutes les machines du réseau local

Les routeurs connectés sur un lien en diffusant à :

- ▷ `FF02::2` tous les routeurs connectés au réseau local
- ▷ `FF02::6` OSPFv3 All DR routers
- ▷ `FF02::9` RIP routers
- ▷ `FF0x::101` Network Time Protocol

Les serveurs DHCP du site :

- `FF02::1:2` All DHCPv6 servers and relay agents on the local network segment (defined in RFC 3315)

Un routeur peut s'annoncer en diffusant vers :

- `FF02::1` groupe composé de tous les noeuds connectés au lien.

Pour utiliser le mDNS, ou «*multicast DNS*» : `FF02:0:0:0:0:0:0:FB`.

La liste des groupes définies peut être consultées à l'adresse
http://www.iana.org/assignments/ipv6-multicast-addresses.

- Abandon des adresses «Site-Local» pour cause de définition insuffisante ;
- définition des adresses ULA «`FC00::/7`» :

```
| 7 bits |1|  40 bits   |  16 bits  |          64 bits           |  P FC00::/7 prefix to
+--------+-+------------+-----------+----------------------------+     identify Local IPv6 unicast
| Prefix |L| Global ID  | Subnet ID |        Interface ID        |
+--------+-+------------+-----------+----------------------------+
```

- but : disposer de réseaux privés «subnettables» à la manière des adresses «*global unicast*» ;
- l'espace d'adresse en «`FC00::/7`» est décomposé en deux sous espaces :
  - ◇ les adresses «`FC00::/8`» non utilisées (le bit L est à 1 ou «*locally assigned*») :
  - ◇ les adresses «`FD00::/8`» utilisées pour fournir un préfixe en «`/48`» :
    - ⋆ 8 bits de préfixe prédéfini : «FD» ;
    - ⋆ 40 bits sont choisis de manière aléatoire pour **garantir l'unicité** (utilisation d'un secret et de la fonction de hachage SHA-1) pour définir le «*Global ID*» ;
    - ⋆ soient 48 bits, ce qui laisse 16 bits pour faire du «subnetting», soit le «*Subnet ID*» ;
    - ⋆ exemple : choix des 40 bits «`e4:8dba:82e1`», ce qui donne un préfixe «`fde4:8dba:82e1::/48`» ce qui permet de définir des sous-réseaux :
      - ▷ de «`fde4:8dba:82e1::/64`»
      - ▷ à «`fde4:8dba:82e1:ffff::/64`».
    - ⋆ le «*Interface ID*» correspond à l'identifiant de l'interface.
- ces réseaux ne peuvent être alloués par un «*Registry*», ils ne sont pas garantis être unique à l'échelle d'Internet et le «reverse DNS» ne peut pointer sur eux ;
- ils sont **routables** suivant le choix des routeurs mais limitation à une organisation : **pas routable globalement**.

Dans la trame Ethernet

Au niveau du type de la trame Ethernet, dans le champs Type :

Type	Protocole
2048 (0x0800)	IPv4
34525 (0x86DD)	IPv6

Au niveau de l'adressage pour le «*broadcast*» :

- ▷ en IPv4, utilisation de l'adresse Ethernet destination `FF:FF:FF:FF:FF:FF` (broadcast du protocole DoD ou Ipv4) ;
- ▷ en IPv6, **plus de broadcast** mais du **multicast** avec les @MAC de `33:33:00:00:00:00` à `33:33:FF:FF:FF:FF`

Plus de broadcast, mais uniquement du multicast pour IPv6, avec des @MAC construites à partir des @IPv6 :

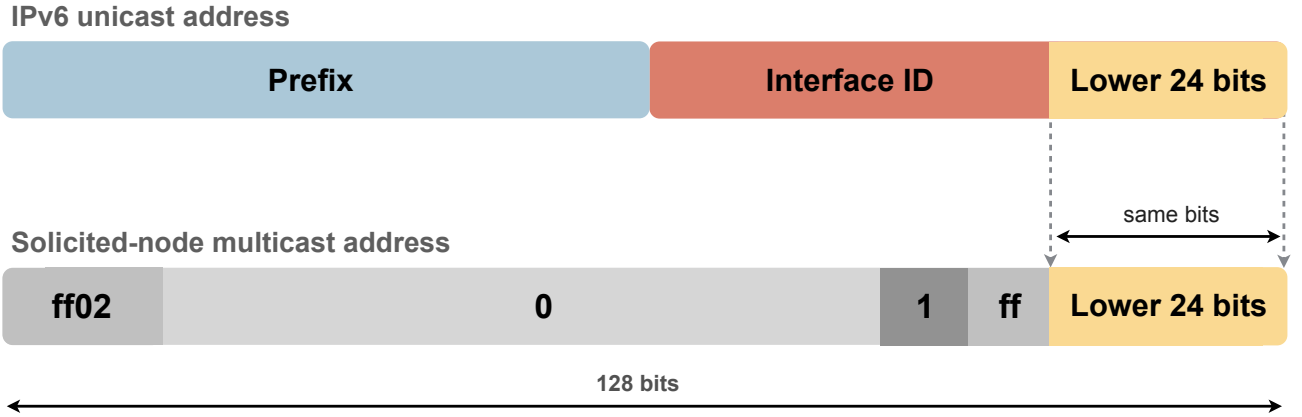
Méhode générale : *on ajoute le préfixe «`33:33`» aux 32 derniers bits de l'@IPv6 de multicast* ;

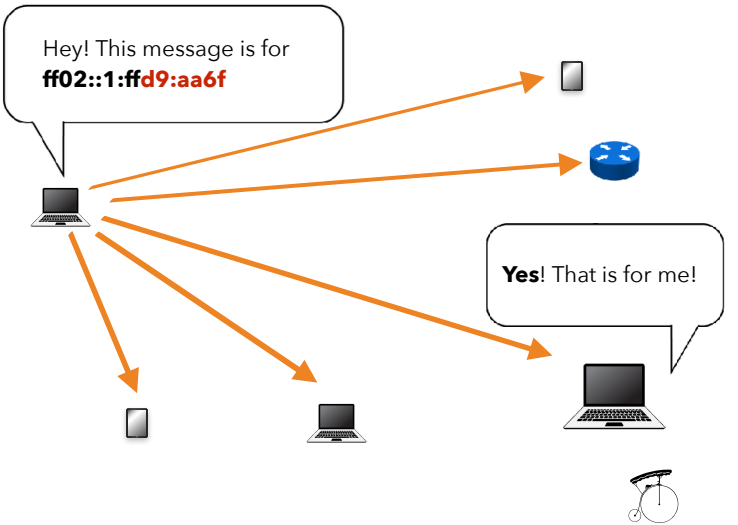
1. pour atteindre un **groupe multicast**, comme par exemple le groupe de toutes les machines du lien local : `FF02::1` ⟹ `33:33:00:00:00:01`

2. pour **atteindre une machine dont on connaît l'adresse IPv6** :
   - ◇ on détermine l'@IPv6 du groupe Multicast de «*Neighbor Sollicitation*» du NDP que l'on verra plus loin :
     - ⋆ le préfixe multicast `FF02:0:0:0:0:1:FF` devant les derniers 24 bits de l'@IPv6 de la cible :
       Exemple : `FE80::02AA:00FF:FE28:9C5A` donne en adresse mutlicast : `FF02::1:FF28:9C5A`
     - ⋆ On construit l'@MAC correspondante à cette @IPv6 multicast (`33:33` + 24 derniers bits de son adresse IPv6) :
       Sur l'exemple : `FF02::1:FF28:9C5A` ⟹ `33:33:FF:28:9C:5A`

   Remarque : cela marche aussi si la machine s'est auto configurée :
   - ◇ Exemple : soit une machine A disposant de l'adresse MAC (OUI) `00:bb:cc:dd:ee:ff`
     - ⋆ Obtention de l'EUI : `02:bb:cc(:FF:FE):dd:ee:ff`
     - ⋆ Obtention de son adresse de lien : `FE80::02bb:ccFF:FEdd:eeff` et de NS : `FF02::1:FFdd:eeff`
     - ⋆ Adresse Ethernet multicast utilisée pour la joindre : `33:33:FF:dd:ee:ff`

   *On espère que l'@MAC de multicast ne touchera que la machine cible ou, au pire, quelques machines mais **pas toutes les machines** connectées au lien local (on évite le broadcast utilisé dans IPv4 qui encombre le réseau).*


IPv6 unicast address
Prefix
Interface ID
Lower 24 bits
same bits
Solicited-node multicast address
ff02
0
1
ff
Lower 24 bits
128 bits


Hey! This message is for
ff02::1:ffd9:aa6f
Yes! That is for me!

On va ajouter une adresse globale unicast `2001::1/64` à l'interface `eth0` d'une machine sous Linux :

xterm
```
root@neogeo:~# ip address add 2001::1/64 dev eth0
```

On vérifie que le routage a bien été mis en place grâce à l'indication du préfixe avec l'adresse `/64` :

xterm
```
root@neogeo:~# ip -6 r
2001::/64 dev eth0  proto kernel  metric 256
```

On déclenche un «ping» vers une machine appartenant à ce réseau pour voir comment va se comporter la machine :

xterm
```
root@neogeo:~# ping6 2001::2
PING 2001::2(2001::2) 56 data bytes
From 2001::1 icmp_seq=1 Destination unreachable: Address unreachable
From 2001::1 icmp_seq=2 Destination unreachable: Address unreachable
```

Ici, il n'y a pas à ajoujer de Zone_ID «`%eth0`» car l'adresse est globale unicast et suffit à identifier l'interface à utiliser.

On sniffe le réseau sur l'interface `eth0` :

xterm
```
pef@neogeo:~$ sudo tcpdump -nvveX -i eth0 ip6
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
18:51:26.959601 00:0c:29:f1:55:61 > 33:33:ff:00:00:02, ethertype IPv6 (0x86dd), length 86: (hlim 255, next-header ICMPv6 (58)
  payload length: 32) 2001::1 > ff02::1:ff00:2: [icmp6 sum ok] ICMP6, neighbor solicitation, length 32, who has 2001::2
      source link-address option (1), length 8 (1): 00:0c:29:f1:55:61
        0x0000:  000c 29f1 5561
    0x0000:  6000 0000 0020 3aff 2001 0000 0000 0000  `.....:.........
    0x0010:  0000 0000 0000 0001 ff02 0000 0000 0000  ................
    0x0020:  0000 0001 ff00 0002 8700 ba39 0000 0000  ...........9....
    0x0030:  2001 0000 0000 0000 0000 0000 0000 0002  ................
    0x0040:  0101 000c 29f1 5561                      ....).Ua
```

On constate que, pour trouver l'@MAC de la machine cible, la machine source utilise le protocole «*ICMP6 neighbor solicitation*» avec l'adresse IPv6 multicast `ff02::1:ff00:2` constituée de l'adresse multicast de solicitation `FF02::1:ff00:0/104` + les derniers 24 bits de l@IPv6 de la cible (ici, `000002`) ; d'où une adresse MAC multicast de destination de la trame constituée du préfixe «`33:33`» avec les 32 derniers bits de l'@IPv6 (ici, `00 00 00 02`).

Pour la diffusion dans un switch

http://en.wikipedia.org/wiki/Multicast_address

Ethernet frames with a value of 1 in the least-significant bit of the first octet of the destination address are treated as multicast frames and are flooded to all points on the network.
While frames with ones in all bits of the destination address (FF:FF:FF:FF:FF:FF) are sometimes referred to as broadcasts, Ethernet network equipment generally does not distinguish between multicast and broadcast frames.
Modern Ethernet controllers filter received packets to reduce CPU load, by looking up the hash of a multicast destination address in a table, initialized by software, which controls whether a multicast packet is dropped or fully received.

Et en IPv6 ?

Le premier octet d'une @MAC de diffusion limitée est `33` et il finit bien par un bit à 1 (impair)... *Ouf !*

Une analyse de paquet en IPv6 avec Scapy

xterm

```
pef@darkstar:~$ sudo tcpdump -c 3 -i en6 -w capture_ipv6 ip6 and tcp
tcpdump: listening on en6, link-type EN10MB (Ethernet), capture size 65535 bytes
3 packets captured
pef@darkstar:~$ scapy
Welcome to Scapy (2.2.0)
>>> rdpcap('capture_ipv6')
>>> lp.summary()
Ether / IPv6 / TCP 2001:41d0:fe2e:700:c57f:7973:1dce:e089:60939 > 2001:41d0:2:eb43::b7f9:7a68:websm S
Ether / IPv6 / TCP 2001:41d0:2:eb43::b7f9:7a68:websm > 2001:41d0:fe2e:700:c57f:7973:1dce:e089:60939 SA
Ether / IPv6 / TCP 2001:41d0:fe2e:700:c57f:7973:1dce:e089:60939 > 2001:41d0:2:eb43::b7f9:7a68:websm A
>>> lp[0]
<Ether  dst=58:98:35:3c:a7:1a src=80:1f:02:7c:db:93 type=0x86dd |<IPv6  version=6L tc=0L fl=0L plen=44
nh=TCP hlim=64 src=2001:41d0:fe2e:700:c57f:7973:1dce:e089 dst=2001:41d0:2:eb43::b7f9:7a68 |
<TCP  sport=60939 dport=websm seq=1979112726 ack=0 dataofs=11L reserved=0L flags=S window=65535
chksum=0x5736 urgptr=0 options=[('MSS', 1220), ('NOP', None), ('WScale', 1), ('NOP', None), ('NOP', None),
('Timestamp', (1012538369, 0)), ('SAckOK', ''), ('EOL', None)] |>>>
>>> hexdump(lp[0])
0000   58 98 35 3C A7 1A 80 1F  02 7C DB 93 86 DD 60 00   X.5<.....|....`.
0010   00 00 00 2C 06 40 20 01  41 D0 FE 2E 07 00 C5 7F   ...,.@ .A.......
0020   79 73 1D CE E0 89 20 01  41 D0 00 02 EB 43 00 00   ys.... .A....C..
0030   00 00 B7 F9 7A 68 EE 0B  23 82 75 F6 DD 16 00 00   ....zh..#.u.....
0040   00 00 B0 02 FF FF 57 36  00 00 02 04 04 C4 01 03   ......W6........
0050   03 01 01 01 08 0A 3C 5A  1C 01 00 00 00 00 04 02   ......<Z........
0060   00 00                                             ..
```


```
>>> i=IPv6()
>>> i.dst="2001:db8:dead::1"
>>> h=IPv6ExtHdrRouting()
>>> h.addresses=["2001:db8:dead::1","2001:db8:dead::1"]
>>> p=ICMPv6EchoRequest()
>>> pa=(i/h/p)
>>> pa
<IPv6  nh=Routing Header dst=2001:db8:dead::1 |
<IPv6ExtHdrRouting  nh=ICMPv6 addresses=[ 2001:db8:dead::1, 2001:db8:dead::1 ] | <ICMPv6EchoRequest  |>>>
>>> pa.show2()
###[ IPv6 ]###
  version= 6L
  tc= 0L
  fl= 0L
  plen= 64
  nh= Routing Header
  hlim= 64
  src= 2001:41d0:fe2e:700:31b2:acde:606a:589e
  dst= 2001:db8:dead::1
###[ IPv6 Option Header Routing ]###
     nh= ICMPv6
     len= 6
     type= 0
     segleft= 3
     reserved= 0L
     addresses= [ 2001:db8:dead::1, 2001:db8:dead::1 ]
###[ ICMPv6 Echo Request ]###
        type= Echo Request
        code= 0
        cksum= 0x74bb
        id= 0x0
        seq= 0x0
        data= ''
>>> hexdump(Ether()/pa)
0000   58 98 35 3C A7 1A 00 26  BB 15 8E C5 86 DD 60 00
0010   00 00 00 40 2B 40 20 01  41 D0 FE 2E 07 00 31 B2
0020   AC DE 60 6A 58 9E 20 01  0D B8 DE AD 00 00 00 00
0030   00 00 00 00 00 01 3A 06  00 03 00 00 00 00 20 01
0040   0D B8 DE AD 00 00 00 00  00 00 00 00 00 01 20 01
0050   0D B8 DE AD 00 00 00 00  00 00 00 00 00 01 20 01
0060   0D B8 DE AD 00 00 00 00  00 00 00 00 00 01 80 00
0070   74 BB 00 00 00 00
```


Un protocole de découverte des voisins, «*Neighbor discovery protocol*» sur le lien local (RFC 2461) :

- ▷ Il remplace le protocole ARP qui n'existe plus en IPv6 ;
- ▷ Il est intégré dans ICMPv6, protocole de niveau 3 ;
  - ◇ *N'utilise plus les capacités de «broadcast» du niveau 2, comme Ethernet* ;
  - ◇ *donne plus de souplesse d'utilisation en particulier sur les réseaux qui ne supportent pas la diffusion*.

Ce protocole réalise les différentes fonctions :

- ◦ la **résolution d'adresses**. Le principe est très proche du protocole ARP que l'on trouve avec IPv4 ;
- ◦ la **détection d'inaccessibilité** des voisins ou NUD, «*Neighbor Unreachability Detection*» ;
- ◦ la **configuration automatique** des équipements connectés au réseau.

Principe

Les nœuds IPv6 qui se trouvent sur le même lien utilise ce protocole pour :

- ▷ découvrir la présence d'autres noeuds,
- ▷ pour déterminer les adresses de liens (choix d'une adresse libre) ;
- ▷ pour trouver les routeurs et pour maintenir les informations concernant les chemins vers les voisins actifs.

Moyens

Au démarrage, chaque noeud Ipv6 joint différents groupes multicast particulier :

- ◦ `FF02::1`, regroupant tous les noeuds d'un même lien ;
- ◦ `FF01::1`, regroupant toutes les applications d'un même nœud ;

Une fois son @IPv6 de lien définie (par autoconfiguration à partir de sa propre @MAC, par tirage aléatoire ou avec DHCPv6) :

- ◦ `FF02::1:FF` + 24 derniers bits de son @IPv6, c-à-d son groupe de «*Neighbor Sollicitation*» pour le protocole Neighbor Discovery Protocol.

Vérification de la configuration d'une pile TCP/IP v6

```
xterm
pef@solaris:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state
UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP qlen 1000
    link/ether 00:11:de:ad:be:ef brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.83/24 brd 192.168.1.255 scope global
eth0
    inet6 fe80::211:deff:fead:beef/64 scope link
       valid_lft forever preferred_lft forever

pef@solaris:~$ netstat -g
IPv6//Adhésions au groupe IPv4
Interface       RefCnt Group
--------------- ------ ---------------------
lo              1      all-systems.mcast.net
eth0            1      224.0.0.251
eth0            1      all-systems.mcast.net
lo              1      ip6-allnodes
eth0            1      ff02::fb
eth0            1      ff02::1:ffad:beef
eth0            1      ip6-allnodes
```

❶⇒l'@MAC de l'interface `eth0` ;

❷⇒l'@IPv6 créée à partir de l'@MAC ;

❸⇒le groupe auquel la machine appartient en rapport avec son @MAC :

- ◇ce groupe se traduit par une adresse MAC particulière : `33:33:ff:ad:be:ef`
- ◇lorsque la machine n'est pas connue par son @MAC, on utilise cette adresse de groupe pour diffuser un message pour la trouver suivant le protocole «*Neighbor Discovery*» : le «*neighbor solicitation*» ;
- ◇la machine «solicitée» répondra alors par un message «*Neighbor advertisement*» .

❹⇒Ce groupe correspond au mDNS, ou en IPv4 le groupe `224.0.0.251` (`FB` = 251).

Adresse multicast de recherche de la machine cible («Neighbor Discovery»)

L'adresse de multicast est `ff02::1:ff00:0/104` avec les 24 derniers bits de l'@IP de la machine cible.

On distingue 5 types de messages :

- **Router Solicitation** : quand leur interface démarre, les noeuds peuvent envoyer ce message afin d'inciter les routeurs à générer des «Router Advertisements» ;
- **Router Advertisement** : les routeurs avertissent de leur présence soit périodiquement, soit après un message de type «Router Solicitation» ;
- **Neighbor Solicitation** : envoyé par un noeud afin d'obtenir l'adresse physique d'un voisin ou de vérifier la disponibilité de celui-ci (utile pour s'assurer de l'unicité de sa propre adresse) ;
- **Neighbor Advertisement** : la réponse au message précédent.
  Un noeud peut aussi envoyer un Neighbor Advertisement afin d'annoncer un changement d'adresse de la couche liaison.
- **Redirect** : utilisé par les routeurs pour informer les noeuds d'un meilleur premier saut pour une destination.

Pour découvrir les routeurs connectés

Sur les liens de multicast, chaque routeur envoie périodiquement un «*Router Advertisement*» annonçant sa disponibilité.
Un noeud présent sur le lien construit ainsi une liste de routeurs par défaut.

Pour découvrir l'adresse Physique d'une autre machine

Un noeud accomplit une résolution d'adresse de niveau 2, en multicastant un «*Neighbor Solicitation*» vers l'adresse multicast du noeud à découvrir.
La cible retourne son adresse de couche liaison dans un «*Neighbor Advertisement*» (unicast).

Ce mécanisme est utilisé par le «*Neighbor Unreachability Detection*» qui permet de tester la présence ou la disparition d'un nøeud, en particulier celle du routeur :

- ▷ une table «Neighbor cache» est construite pour mémoriser les associations (@MAC,@IP) ;
- ▷ chaque entrée possède un état : «`INCOMPLETE`», «`REACHABLE`», «`STALE`», «`DELAY`», «`PROBE`» :
  - ◇ l'état «`INCOMPLETE`» indique que l'on démarre une recherche de présence ;
  - ◇ l'état «`REACHABLE`» indique la présence de l'interlocuteur : réception de paquets provenant de lui ;
  - ◇ celui «`STALE`» indique un état inconnu et une mise en tampon des paquets à destination du nœud avant de passer dans l'état «`DELAY`» ;
  - ◇ l'état «`DELAY`» déclenche une mise en attente (attente d'un paquet en retour du nøeud par exemple) avant un passage dans l'état «`PROBE`» ;
  - ◇ l'état «`PROBE`» indique l'envoi d'un paquet «Neighbor solicitation» et sans réponse les paquets à destination sont détruits et une erreur est déclenchée.

L'avantage de ce mécanisme est d'utiliser des messages ICMPv6 explicite pour connaitre l'état mais aussi de surveiller le trafic existant pour connaître cet état (paquets TCP, UDP etc. *en provenance du nœud surveillé.*
L'envoi d'un ping vers «`ff02::1`» permet de remplir sa table de voisinage.

Avantages du Neighbor discovery protocol

En IPv4, les mécanismes offerts par IPv6 sont implémentés grâce :

- ▷ au protocole ARP ;
- ▷ au type de message ICMP «*Router Discovery*» ;
- ▷ au type de message ICMP «*Redirect*».

Cependant en IPv4, il n'existe pas de moyen qui permette la détection de l'indisponibilité d'un noeud.

Le «*Neighbor discovery*» apporte des améliorations :

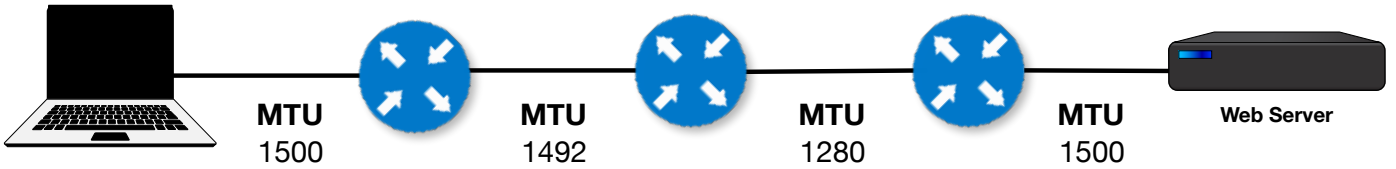
- □ le «*Router Discovery*» est incorporé dans IPv6 (il utilise ICMPv6) ;
- □ un «*Router Advertisement*» transporte :
  - ◇ l'**adresse de la couche liaison** de l'émetteur, aucun paquet supplémentaire n'est nécessaire pour la résoudre ;
  - ◇ la liste des **préfixes utilisables** ;
  - ◇ une valeur de «Max Hop Limit» ;
  - ◇ la **durée de validité** de ces préfixes (utile pour faire migrer un réseau en fonctionnement !) ;
  - ◇ la **MTU** du lien ;
- □ les «*Router Advertisements*» permettent de faire de l'**autoconfiguration** d'adresse ;
- □ la détection «d'indisponibilité d'un voisin» améliore le taux de paquets délivrés en cas de routeurs défectueux ;
- □ le protocole permet de déterminer si l'adresse que l'on s'est choisi est **déjà prise** (autoconfiguration) ;
- □ le fait de réaliser la résolution d'adresse en utilisant ICMPv6 rend le protocole plus indépendant du média que ARP et rend possible l'utilisation de mécanismes **d'authentification** et **de sécurité** IP.

Le «*Path MTU Discovery*»

Principe

- Le «Path» ou chemin est l'ensemble des liens traversés par un paquet IPv6 entre la source et la destination.
- Le link MTU = taille maximale de l'unité de transmission en octets qui peut être transportée sans fragmentation sur le lien.
- Le Path MTU (ou pMTU) est le plus petit { link MTU } pour un chemin donné.
- Le « *Path MTU Discovery*» permet la découverte automatique du pMTU pour un chemin donné.

Mise en oeuvre



On utilise le protocole ICMPv6 :

1. on envoie au destinataire un paquet ICMPv6 le plus gros possible ;
2. il renvoie un message «*Packet size too large*» si le datagramme ne peut pas passer au travers d'un lien avec une indication sur la taille à utiliser.

- similaire au «*IGMP snooping*», «*Internet Group Management Protocol*» qui est réalisé par le switch pour déterminer sur quel port relayer une trame envoyée en multicast (au lieu de l'envoyer en broadcast sur l'ensemble des ports) :
  - une machine indique son entrée dans un « groupe multicast» en envoyant un paquet IGMP sur le port du switch auquel elle est connectée (ce paquet est lui-même envoyé en multicast vers un groupe prédéfini) ;
  - le switch «*écoute*» ce paquet et inclus le port dans la liste des destinataires pour ce groupe multicast ;
  - la machine peut également notifier le switch de sa sortie du groupe multicast.

- dérivé d'IGMP, IGMPv3, RFC 3376, $\Longrightarrow$ MLDv2, RFC3810, et intégré dans les paquets ICMPv6.
  Exemple : l'ajout d'une adresse sur l'interface :

xterm
```
$ sudo ip a add dev wlp1s0 2001::1/64
```

produit l'envoi d'un «*Multicast Listener Report v2*» pour le groupe multicast dérivé de cette adresse `ff02::1:ff00:1`

xterm
```
$ sudo tcpdump -i wlp1s0 -nvveX ip6
10:58:18.545615 78:92:9c:e4:5b:83 > 33:33:ff:00:00:01, ethertype IPv6 (0x86dd), length 86: (hlim
255, next-header ICMPv6 (58) payload length: 32) :: > ff02::1:ff00:1: [icmp6 sum ok] ICMP6, neighbor
solicitation, length 32, who has 2001::1

10:58:17.933442 78:92:9c:e4:5b:83 > 33:33:00:00:00:16, ethertype IPv6 (0x86dd), length 190: (hlim 1,
next-header Options (0) payload length: 136) fe80::f00f:b8f2:14b8:125 > ff02::16: HBH (rtalert:
0x0000) (padn) [icmp6 sum ok] ICMP6, multicast listener report v2, 6 group record(s) [gaddr
ff02::1:ff00:1 to_ex { }] [gaddr ff02::fb to_ex { }] [gaddr ff02::1:ff94:1e58 to_ex { }] [gaddr
ff02::1:ff5a:fd23 to_ex { }] [gaddr ff02::1:ffb8:125 to_ex { }] [gaddr ff12::8384 to_ex { }]
```

Le premier paquet de «neighbor solicitation» *sert à s'assurer que l'adresse n'est pas déjà utilisée par un autre poste avant de l'associer à l'interface.*

On rejoint le groupe `ff02::1:ff1a:ef1e` avec `socat` :

xterm

```
$ socat stdio udp6-recvfrom:7182,ipv6-add-membership=❷'[ff02::1:ff1a:ef1e%eth0]':eth0
```

Ce qui déclenche l'envoi vers le groupe `ff02::16` de l'entrée dans le groupe multicast :

xterm

```
$ sudo tcpdump -lnvvX -i toto
tcpdump: listening on toto, link-type EN10MB (Ethernet), capture size 262144 bytes
16:52:45.440506 IP6 (hlim 1, next-header Options (0) payload length: 36) fe80::203:ffff:fe18:cf1e >
ff02::16: HBH (rtalert: 0x0000) (padn
) [icmp6 sum ok] ICMP6, multicast listener report v2, 1 group record(s) [gaddr ff02::1:ff1a:ef1e to_ex
{ }]
        0x0000:  6000 0000 0024 0001 fe80 0000 0000 0000  `....$..........
        0x0010:  0203 ffff fe18 cf1e ff02 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0016 3a00 0502 0000 0100  ........:.......
        0x0030:  8f00 b295 0000 0001 0400 0000 ff02 0000  ................
        0x0040:  0000 0000 0000 0001 ff1a ef1e            ............
16:52:46.092397 IP6 (hlim 1, next-header Options (0) payload length: 36) fe80::203:ffff:fe18:cf1e >
ff02::16: HBH (rtalert: 0x0000) (padn
) [icmp6 sum ok] ICMP6, multicast listener report v2, 1 group record(s) [gaddr ff02::1:ff1a:ef1e to_ex
{ }]
        0x0000:  6000 0000 0024 0001 fe80 0000 0000 0000  `....$..........
        0x0010:  0203 ffff fe18 cf1e ff02 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0016 3a00 0502 0000 0100  ........:.......
        0x0030:  8f00 b295 0000 0001 0400 0000 ff02 0000  ................
        0x0040:  0000 0000 0000 0001 ff1a ef1e            ............
```

xterm

```
$ ip maddr
7:      eth0
        link  01:00:5e:00:00:01
        link  33:33:00:00:00:01
        link  ❷33:33:ff:1a:ef:1e
        inet  224.0.0.1
        inet6 ff02::1:ff1a:ef1e
        inet6 ff02::1
        inet6 ff01::1
```

L'adresse MAC ❷ est liée au groupe ❶.

Le groupe multicast `ff02::16` correspond au groupe `All MLDv2-capable routers`, ce qui permet à ces routeurs de savoir que la machine a rejoint le groupe `ff02::1:ff1a:ef1e`.

La configuration automatique signifie qu'une machine obtient **toutes les informations nécessaires** à sa connexion à un réseau local IP **sans aucune intervention humaine**.

L'auto-configuration d'adresses sous IPv6 a pour objectif :

– l'**acquisition d'une adresse** quand une machine est attachée à un réseau pour la première fois ;
– l'**obtention de la nouvelle adresse** suite à la renumérotation des machines du site après un changement d'opérateur (changement de FAI) ;

Le processus d'auto-configuration d'adresse d'IPv6 comprend :

- la **création d'une adresse** lien-local ;
- la **vérification** de son unicité ;
- la **détermination** des adresses unicast globales (obtention du préfixe).

IPv6 spécifie deux méthodes d'auto-configuration pour l'adresse unicast globale :

▷ l'auto-configuration **sans état**, «*stateless auto-configuration*», (RFC2462) :
L'auto-configuration sans état ne demande aucune configuration manuelle des machines, une configuration minimum pour les routeurs et aucun serveur supplémentaire :
  ◇ Elle se sert du protocole ICMPv6 et peut fonctionner sans la présence de routeurs ;
  ◇ Elle nécessite un sous-réseau à diffusion ;
  Cette méthode ne s'applique que pour les machines et pas pour la configuration des routeurs.
  ◇ Une machine génère son adresse IPv6 à partir d'infos locales (@MAC par ex.) et d'infos fournies par un routeur (préfixe).
  ◇ Le routeur fournit à la machine les infos sur le sous-réseau associé au lien, il donne les **préfixes**.

▷ l'auto-configuration **avec état**, «*stateful auto-configuration*», DHCPv6 :
  ◇ offre une information de configuration plus riche et un contrôle sur l'affectation des paramètres de configuration (adressage, routage, DNS, NIS, . . .) ;
  ◇ utilise le modèle du client-serveur :
    ⋆ le client est la machine candidate à une connectivité globale IPv6 ;
    ⋆ le serveur DHCP peut ne pas être sur le même lien (sous-réseau) et peut nécessiter de passer par un relais.

Autoconfiguration sans état : SLAAC, «*Stateless Address Autoconfiguration*»

- SLAAC est fourni par le «*Neighbor Discovery Protocol*» ;
- «stateless» ou «sans état» : choix d'adresse interface autonome :
  - pas de serveur DHCPv6 : il n'y a pas de «*pool*» d'adresses fournies et gérées par le serveur DHCP («*leasing*») ;
  - **mais** s'il y en a un, il peut fournir des informations «stateless» comme les adresses des serveurs DNS ;
- Utilisation du «*router solicitation*» permet d'obtenir un **préfixe réseau globale** :
  - envoyé à destination du groupe des routeurs :
    - adresse IPv6 «`ff02::2`» ;
    - adresse MAC : «`33:33:00:00:00:02`»
  - depuis :
    - adresse IPv6 «*link-local*» de l'interface du matériel de préfixe «FE80::/64» associé au 64 bits d'adresses MAC EUI ;
    - adresse MAC de l'interface.
  - **si** :
    - r**outeur présent** répond avec les **préfixes réseau disponible**s et les **passerelles** à utiliser (lui-même éventuellement) ;
    - **pas de routeur présent** : relance de la découverte puis abandon si toujours pas de réponse ;
    - **plusieurs routeurs possibles** : choix du meilleur à l'aide du «*Neighbor Unreachability Detection*»
  - définition de **deux adresses globales** :
    - une temporaire aléatoire, dont on vérifie l'unicité grâce au «*Duplicate Address Detection*», c-à-d par l'envoi préalable d'un «*Neighbor Solicitation*» vers cette adresse :

```
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:fd:cf:f7 brd ff:ff:ff:ff:ff:ff
    inet6 2001:789d:bebe:900:5597:92f1:5880:2ec0/64 scope global temporary dynamic
       valid_lft 294sec preferred_lft 114sec
```

    - une liée à l'adresse MAC de l'interface :

```
    inet6 2001:789d:bebe:900:a00:27ff:fefd:cff7/64 scope global dynamic
       valid_lft 294sec preferred_lft 114sec
```


Ajout d'une adresse IPv6 pour une interface du container

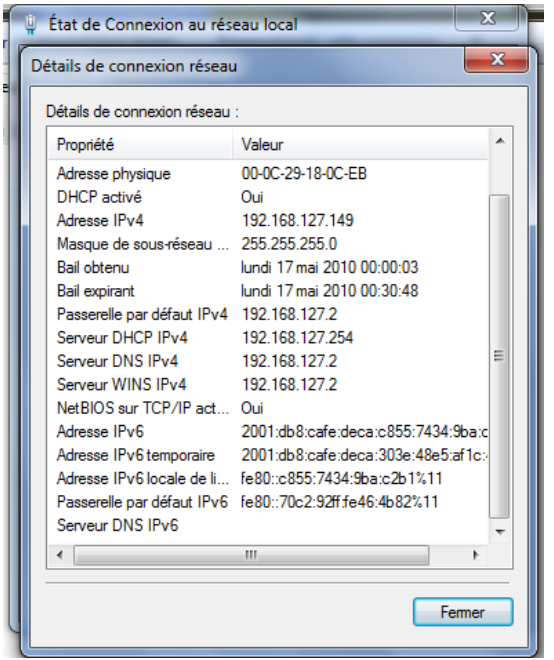
Dans la configuration d'un container, juste après la définition d'une interface :

```
1  lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596/64
```

Pour finir le transparent… IPv6 sous GNU/Linux et Windows 7

```
ubuntu@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:22:ca:2f
          inet addr:192.168.127.128  Bcast:192.168.127.255  Mask:255.255.255.0
          inet6 addr: 2001:db8:cafe:deca:20c:29ff:fe22:ca2f/64 Scope:Global
          inet6 addr: fe80::20c:29ff:fe22:ca2f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:285 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39418 (38.4 KB)  TX bytes:3706 (3.6 KB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```



- **Espace d'adressage** considérablement augmenté ;
- **Routage étendu** :
  - plus de niveau hiérarchique d'adressage ;
  - auto-configuration simplifiée des adesses ;
- Gestion améliorée du **routage multicast** et **abandon du broadcast** :
  - moins d'encombrement du switch ;
  - meilleure performance du réseau ;
- **Entête simplifiée** :
  - moins de champs comparé à IPv4 ;
  - moins de temps de traitement nécessaire pour les traiter ;
  - champs supprimés placés dans des entêtes optionnels.
- Entêtes **optionnelles** :
  - plus rapide à traiter par les routeurs, car traitées seulement par le destinataire ;
  - des entêtes qui peuvent grossir au besoin ;
- Possibilité de faire de **l'authentification** et de la **confidentialité** avec du chiffrement ;
- Possibilité de faire du **routage par la source** ;
- Possibilité de **Qualité de Service** basée sur les «*Differentiated Services*».

Le client

```
#!/usr/bin/python

import socket

adresse_serveur = 'fe80::211:deff:fead:beef'
numero_port_serveur = 8080
flow_info = 0
scope_id = 0

ma_socket = socket.socket(socket.AF_INET6,socket.SOCK_STREAM)
# flow_info et scope_id peuvent être omis
ma_socket.connect((adresse_serveur,numero_port_serveur,flow_info,scope_id))

ma_socket.close()
```

Le serveur

```
#!/usr/bin/python

import socket

adresse_serveur = '::'
numero_port_serveur = 8080

ma_socket = socket.socket(socket.AF_INET6,socket.SOCK_STREAM)
ma_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
ma_socket.bind((adresse_serveur, numero_port_serveur))
ma_socket.listen(socket.SOMAXCONN)

(nouvelle_connexion,depuis) = ma_socket.accept()
print "Connexion depuis ", depuis
nouvelle_connexion.close()
```


Le ping

xterm
```
pef@solaris:~$ sudo ip neigh flush dev eth0
pef@solaris:~$ ping6 -c 1 -I eth0 fe80::20c:29ff:fe84:34c4
PING fe80::20c:29ff:fe84:34c4(fe80::20c:29ff:fe84:34c4) from fe80::211:deff:fead:beef eth0: 56 data bytes
64 bytes from fe80::20c:29ff:fe84:34c4: icmp_seq=1 ttl=64 time=3.14 ms

--- fe80::20c:29ff:fe84:34c4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.148/3.148/3.148/0.000 ms
```

Les routes

xterm
```
pef@solaris:~$ ip -6 route
fe80::/64 dev eth0  proto kernel  metric 256
```

Le «voisinage»

xterm
```
pef@solaris:~$ ip neigh
fe80::20c:29ff:fe84:34c4 dev eth0 lladdr 00:0c:29:84:34:c4 REACHABLE
192.168.1.53 dev eth0 lladdr 00:10:75:1a:d4:8a STALE
192.168.1.254 dev eth0 lladdr 58:98:35:3c:a7:1a STALE
```

Pour sniffer…

xterm
```
# tcpdump -i eth0 -vv ip6
```

Pour faire une requête DNS

xterm
```
dig www.google.com AAAA
```

Pour activer le routage des paquets IPv6

xterm
```
sudo sysctl -w net.ipv6.conf.all.forwarding=1
```

Le «neighbor discovery» illustré

xterm

```
pef@solaris:~$ sudo tcpdump -i eth0 -nvvXXe ip6
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size
65535 bytes
10:50:35.472416 ❶00:11:de:ad:be:ef > ❷33:33:ff:84:34:c4 , ethertype IPv6
(0x86dd),
length 86: (hlim 255, next-header ICMPv6 (58) payload length: 32)
fe80::211:deff:fead:beef > ff02::1:ff84:34c4: [icmp6 sum ok] ICMP6,
neighbor solicitation, length 32, who has fe80::20c:29ff:fe84:34c4
     source link-address option (1), length 8 (1): 00:11:de:ad:be:ef
       0x0000:  0011 dead beef
   0x0000:  3333 ff84 34c4 0011 dead beef 86dd 6000  33..4.........`.
   0x0010:  0000 0020 3aff fe80 0000 0000 0000 0211  ....:...........
   0x0020:  deff fead beef ff02 0000 0000 0000 0000  ................
   0x0030:  0001 ff84 34c4 8700 aba3 0000 0000 fe80  ....4...........
   0x0040:  0000 0000 0000 020c 29ff fe84 34c4 0101  ........)...4...
   0x0050:  0011 dead beef                          .....
10:50:35.473274 ❸00:0c:29:84:34:c4  > 00:11:de:ad:be:ef, ethertype IPv6
(0x86dd),
length 86: (hlim 255, next-header ICMPv6 (58) payload length: 32)
fe80::20c:29ff:fe84:34c4 > fe80::211:deff:fead:beef: [icmp6 sum ok]
ICMP6,
neighbor advertisement, length 32, tgt is fe80::20c:29ff:fe84:34c4,
Flags [solicited, override]
     destination link-address option (2), length 8 (1):
00:0c:29:84:34:c4
       0x0000:  000c 2984 34c4
   0x0000:  0011 dead beef 000c 2984 34c4 86dd 6000  ........).4...`.
   0x0010:  0000 0020 3aff fe80 0000 0000 0000 020c  ....:...........
   0x0020:  29ff fe84 34c4 fe80 0000 0000 0000 0211  )...4...........
   0x0030:  deff fead beef 8800 5e75 6000 0000 fe80  ........^u`.....
   0x0040:  0000 0000 0000 020c 29ff fe84 34c4 0201  ........)...4...
   0x0050:  000c 2984 34c4                          ..).4.
```

La machine `fe80::211:deff:fead:beef` veut contacter la machine `fe80::20c:29ff:fe84:34c4` dont elle ignore l'@MAC : elle réalise un «*neighbor discovery*» :

- le «*Neighbor Discovery*» est un message spécial du protocole ICMPv6 ;
- le paquet est envoyé vers **l'@MAC du groupe** auquel devrait appartenir la machine cible : `33:33:ff:84:34:c4` ❷ ;
- la **machine cible** répond en fournissant ainsi son @MAC réelle `00:0c:29:84:34:c4` ❸.

On peut noter que la machine `fe80::211:deff:fead:beef` a déduit son adresse depuis son adresse MAC ❶.

xterm

```
10:50:35.473285 00:11:de:ad:be:ef > 00:0c:29:84:34:c4, ethertype IPv6
(0x86dd),
length 118: (hlim 64, next-header ICMPv6 (58) payload length: 64)
fe80::211:deff:fead:beef > fe80::20c:29ff:fe84:34c4: [icmp6 sum ok]
ICMP6,
echo request, length 64, seq 1
   0x0000:  000c 2984 34c4 0011 dead beef 86dd 6000  ..).4.........`.
   0x0010:  0000 0040 3a40 fe80 0000 0000 0000 0211  ...@:@..........
   0x0020:  deff fead beef fe80 0000 0000 0000 020c  ................
   0x0030:  29ff fe84 34c4 8000 26a6 54a0 0001 6bb9  )...4...&.T...k.
   0x0040:  274f 842d 0700 0809 0a0b 0c0d 0e0f 1011  'O.-............
   0x0050:  1213 1415 1617 1819 1a1b 1c1d 1e1f 2021  ..............!
   0x0060:  2223 2425 2627 2829 2a2b 2c2d 2e2f 3031  "#$%&'()*+,-./01
   0x0070:  3233 3435 3637                           234567
10:50:35.473552 00:0c:29:84:34:c4 > 00:11:de:ad:be:ef, ethertype IPv6
(0x86dd),
length 118: (hlim 64, next-header ICMPv6 (58) payload length: 64)
fe80::20c:29ff:fe84:34c4 > fe80::211:deff:fead:beef: [icmp6 sum ok]
ICMP6,
echo reply, length 64, seq 1
   0x0000:  0011 dead beef 000c 2984 34c4 86dd 6000  ........).4...`.
   0x0010:  0000 0040 3a40 fe80 0000 0000 0000 020c  ...@:@..........
   0x0020:  29ff fe84 34c4 fe80 0000 0000 0000 0211  )...4...........
   0x0030:  deff fead beef 8100 25a6 54a0 0001 6bb9  ........%.T...k.
   0x0040:  274f 842d 0700 0809 0a0b 0c0d 0e0f 1011  'O.-............
   0x0050:  1213 1415 1617 1819 1a1b 1c1d 1e1f 2021  ..............!
   0x0060:  2223 2425 2627 2829 2a2b 2c2d 2e2f 3031  "#$%&'()*+,-./01
   0x0070:  3233 3435 3637                           234567
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

Le «*ping*» d'IPv6 correspond à l'envoi d'un «*echo request*» entraînant la réception d'un «*echo reply*».

Apprendre la composition d'un réseau

Difficulté à trouver l'adresse des machines :

- ⋆ scanner un réseau en IPv4 peut prendre quelques heures : 2^8 à 2^{16} adresses en général ;
- ⋆ scanner un réseau en IPv6 peut prendre plusieurs années : 2^{64} adresses !

Mais un simple `ping6 ff02::1%eth0` permet de révéler les machines connectées au réseau local...

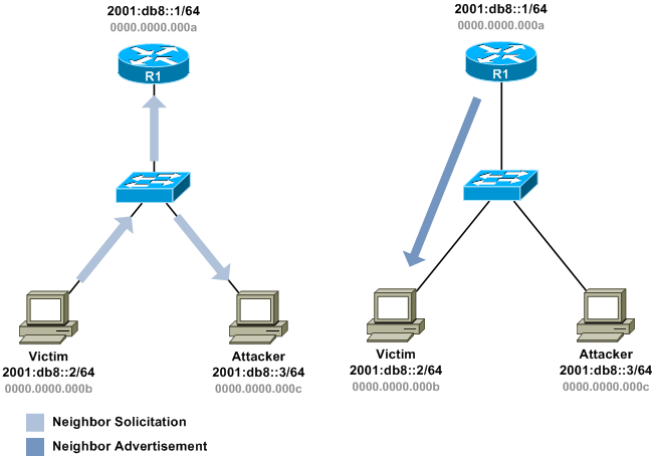
Déterminer la nature d'un OS à partir des méthodes d'IPv4

- ▷ IPv4 : utiliser l'IPID, c-à-d l'identifiant du datagramme IP :
  - ◇ IPv4 : est associé à la fragmentation du datagramme IP (avec les drapeaux et le «fragment offset») ;
  - ◇ IPv6 : n'est plus manipulé par les routeurs mais par la source sous IPv6 ;
  - ◇ IPv6 : le chevauchement de fragment, *overlapping*, est interdit (RFC 5722) ;
  - ◇ *peut servir à identifier l'OS de la machine source* (le drapeau DF a disparu).
  - ◇ *peut être généré de manière aléatoire ou globalement par la machine*.
- ▷ IPv4 : la valeur de départ du TTL :
  - ◇ IPv4 : dépend de l'OS (64, 32 ou 128) ;
  - ◇ IPv6 : transformé en «hop limit» :
    - ⋆ dépend de la valeur transmise par le routeur dans les «router advertisements» (préfixe, MTU et hop limit) ;
      Tous les OS utilisent la même valeur : pas de différentiation.
    - ⋆ dépend de l'OS s'il n'y a pas de routeur présent dans le réseau local ;
      peut être utilisé pour identifier l'OS.
- ▷ IPv4 : «*ICMP port unreachable*», lorsqu'un port ne peut être contacté, la machine renvoie un message ICMP d'erreur reprenant une partie du datagramme d'origine ;
  - ◇ IPv6 : repris en l'état ;
    peut servir pour l'OS fingerprinting.

Exploiter les nouvelles entêtes d'IPv6, le «*Path MTU discovery*», le «*NDP*», *etc.*

MiTM au travers du protocole «Neighbor Discovery»

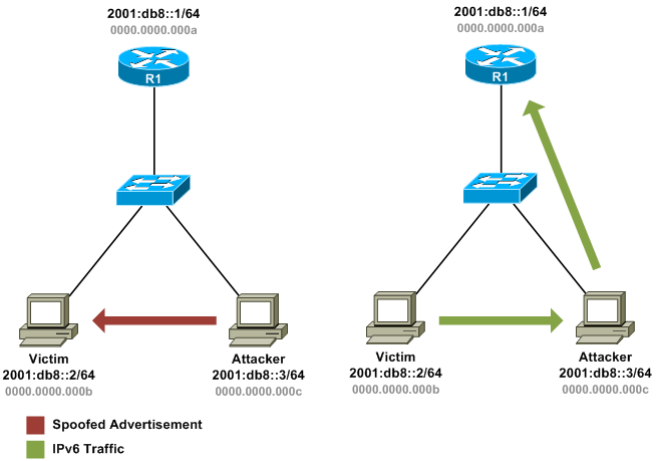
Le but de cette attaque est de **prendre la place du routeur**.



Déroulement de l'attaque :

- l'attaquant répond **à la place du routeur** par un «*Neighbor advertisement*» ;
- l'attaquant relais ensuite les paquets vers le routeur.

- pour connaitre l'@MAC du routeur, la victime envoie un message «*neighbor solicitation*» ;
- ce message et diffusé vers toutes les machines, y compris celle de l'attaquant ;
- le routeur répond par un message «*Neighbor advertisement*» :



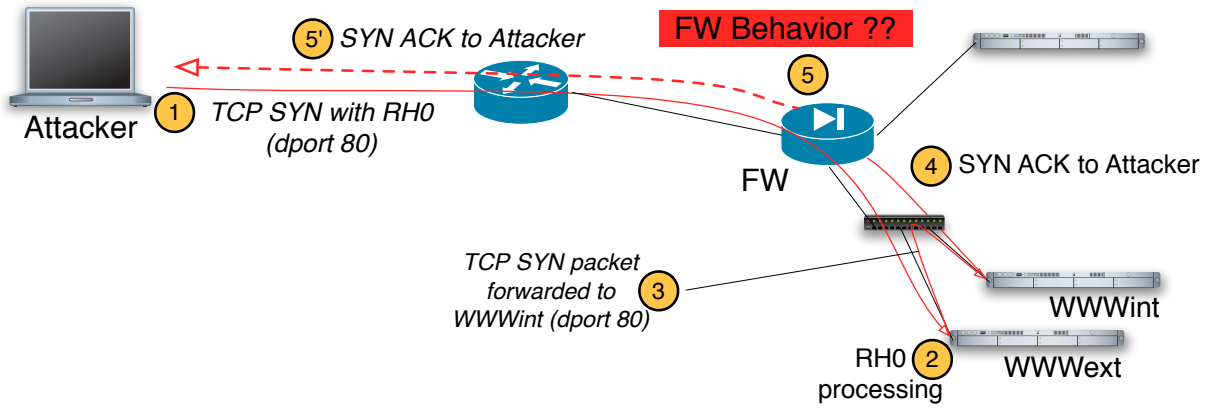
Est-ce difficile à faire ? Un peu de Scapy !

```
Welcome to Scapy (2.1.0)
>>> ls(IPv6
... )
version    : BitField             = (6)
tc         : BitField             = (0)
fl         : BitField             = (0)
plen       : ShortField           = (None)
nh         : ByteEnumField        = (59)
hlim       : ByteField            = (64)
src        : SourceIP6Field       = (None)
dst        : IP6Field             = ('::1')
>>>
>>> ls(IPv6)
version    : BitField             = (6)
tc         : BitField             = (0)
fl         : BitField             = (0)
plen       : ShortField           = (None)
nh         : ByteEnumField        = (59)
hlim       : ByteField            = (64)
src        : SourceIP6Field       = (None)
dst        : IP6Field             = ('::1')
>>> ipv6=IPv6(src='fe80::1', dst='fe80::2')
>>> ipv6
<IPv6  src=fe80::1 dst=fe80::2 |>
>>> ls(ICMPv6ND_NA)
type       : ByteEnumField        = (136)
code       : ByteField            = (0)
cksum      : XShortField          = (None)
R          : BitField             = (1)
S          : BitField             = (0)
O          : BitField             = (1)
res        : XBitField            = (0)
tgt        : IP6Field             = ('::')
>>> na=ICMPv6ND_NA(tgt='fe80::1', R=0)
>>> na
<ICMPv6ND_NA  R=0 tgt=fe80::1 |>
```

$\Longrightarrow$

```
>>> ls(ICMPv6NDOptDstLLAddr)
type       : ByteField            = (2)
len        : ByteField            = (1)
lladdr     : MACField             = ('00:00:00:00:00:00')
>>> ls(ICMPv6NDOptDstLLAddr)
type       : ByteField            = (2)
len        : ByteField            = (1)
lladdr     : MACField             = ('00:00:00:00:00:00')
>>> lla=ICMPv6NDOptDstLLAddr(lladdr='00:00:00:00:00:0c')
>>> lla
<ICMPv6NDOptDstLLAddr  lladdr=00:00:00:00:00:0c |>
>>> paquet=ipv6/na/lla
>>> paquet
<IPv6  nh=ICMPv6 hlim=255 src=fe80::1 dst=fe80::2 |<ICMPv6ND_NA  R=0 tgt=fe80::1
 |<ICMPv6NDOptDstLLAddr  lladdr=00:00:00:00:00:0c |>>>
>>> paquet.show()
###[ IPv6 ]###
  version= 6
  tc= 0
  fl= 0
  plen= None
  nh= ICMPv6
  hlim= 255
  src= fe80::1
  dst= fe80::2
###[ ICMPv6 Neighbor Discovery - Neighbor Advertisement ]###
     type= Neighbor Advertisement
     code= 0
     cksum= None
     R= 0
     S= 0
     O= 1
     res= 0x0
     tgt= fe80::1
###[ ICMPv6 Neighbor Discovery Option - Destination Link-Layer Address ]###
        type= 2
        len= 1
        lladdr= 00:00:00:00:00:0c
>>>
```


Contournement du firewall avec l'extension «*Routing Header*»



`http://www.secdev.org/conf/IPv6_RH_security-csw07.pdf`

$\Longrightarrow$ RFC 5095 : «Deprecation of Type 0 Routing Headers in IPv6»

L'extension «Routing Header» qui permet de contrôler le routage d'un paquet par la source a été supprimé dans sa version «Type 0».

```
Routing types
Due to the fact that with Routing Header type 0 a simple but effective[10] denial-of-service attack could be launched,
this header is deprecated[11] and host and routers are required to ignore these headers.
Routing Header type 1 is used for the Nimrod[12] project funded by DARPA.
Routing Header type 2 is a limited version of type 0 and is used for Mobile IPv6, where it can hold the Home Address
of
the Mobile Node.
```