# 42 PROJECT
## Ft_ping

VOIR :
https://www.geeksforgeeks.org/ping-in-c/
https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/
http://www.microhowto.info/howto/send_an_arbitrary_ethernet_frame_using_an_af_packet_socket_in_c.html

## I. RAPPELS SOCKETS

### 1. *Sockets*

Reprenons, avant de commencer, les bases des sockets dans des systèmes UNIX. Un socket est un *endpoint* de communication, décrit par un *file descriptor*.

Comme on le sait, on obtient un socket par le biais de la fonction `socket` :
`int socket(int domain, int type, int protocol)`

> Le premier argument désigne le *communication domain*, en d'autres termes le mode de communication, la famille de protocole qui sera utilisée pour la communication. AF_UNIX indique un protocol de communication local entre sockets unix, AF_INET indique l'utilisation d'adresses IPv4 pour la communication, AF_INET6 d'adresses IPv6, AF_PACKET indique une communication par le biais de paquets bruts (raw), etc...

> Le second argument désigne le type de communication (une fois choisie la famille de protocole utilisée), c'est-à-dire la sémantique de la communication. On peut avoir notamment les types suivants :

```
SOCK_STREAM
        Provides sequenced, reliable, two-way, connection-based
        byte streams.  An out-of-band data transmission mechanism
        may be supported.

SOCK_DGRAM
        Supports datagrams (connectionless, unreliable messages of
        a fixed maximum length).

SOCK_SEQPACKET
        Provides a sequenced, reliable, two-way connection-based
        data transmission path for datagrams of fixed maximum
        length; a consumer is required to read an entire packet
        with each input system call.

SOCK_RAW
        Provides raw network protocol access.

Etc...
```

Chaque famille de protocole n'implémente pas forcément l'intégralité des types définis.

> Le troisième argument représente le type de protocole à utiliser (une fois la famille de communication et le type de communication définis). En général, on place cet argument à **0** :

"*Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0*".

Bref, la plupart du temps cette fonction sera utilisée pour obtenir une socket AF_INET (communication entre différentes machines via IP), en SOCK_STREAM (protocole TCP) ou en SOCK_DGRAM (protocole UDP) :

```
sd = socket(AF_INET, SOCK_STREAM,0)
sd = socket(AF_INET, SOCK_DGRAM, 0)
```

2. *Connection-oriented socket usage*

Lorsqu'on crée un socket de type SOCK_STREAM, il s'agit d'un socket qui va utiliser le protocole TCP, c'est-à-dire établissant une connexion bidirectionnelle stable entre le socket et sa destination. Ce type de connexion nécessite que le socket SOCK_STREAM soit **connecté** à sa destination, pour commencer à échanger des données avec elle, lui demander de confirmer la réception des données qui lui sont envoyées, etc...

On dispose de deux moyens pour connecter un socket SOCK_STREAM à sa destination.

> Mode client (machine --> destination)

Le premier est d'utiliser la fonction `connect` pour demander à une machine distante d'établir une connection avec nous.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Le second argument *addr* contient ici l'adresse IP et le port de la machine à laquelle on souhaite se connecter et établir une connexion TCP (on utilisera notre IP et un port source aléatoire de notre côté).

C'est ce qu'on a fait dans **Durex** (implémentation reverse shell) :

```
struct sockaddr_in sa;
int                s;

sa.sin_family = AF_INET;
sa.sin_addr.s_addr = (client->sockaddr).sin_addr.s_addr;
sa.sin_port = htons(port);

s = socket(AF_INET, SOCK_STREAM, 0);
connect(s, (struct sockaddr *)&sa, sizeof(sa));
```

> Mode serveur (machine <-- destination)

Au contraire de l'exemple précédent, on va cette fois établir des connexions TCP en acceptant des demandes de connexion.

Pour cela, on utilise l'enchaînement suivant (bind – listen – accept) :

```
int sd = socket(AF_INET, SOCK_STREAM, 0);

sockaddr.sin_family = AF_INET;
sockaddr.sin_port = htons(4242);
sockaddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(sd, (struct sockaddr *)&sockaddr, sizeof(sockaddr));

listen(sd, 50);
```

[...]

```
int new_sd = accept(sd, NULL, NULL)
```

D'abord,, l'appel à **bind** associe notre socket TCP à une IP et un port local (ici tous les interfaces, et le port 4242). Ensuite, l'appel à **listen** place le socket en mode LISTEN (on ouvre le port, et le socket TCP attend des demandes de connexion). Enfin, l'appel à **accept**[1] permet d'accepter les demandes de connexion reçues sur le *listen* socket, et crée un nouveau socket **connecté** à sa destination.

Notons que pour se connecter à notre *listen* socket, le client a probablement utilisé un call à **connect** comme on l'a montré ci-dessus.

Quoi qu'il en soit, qu'on ait utilisé le mode client ou serveur, on dispose désormais d'un socket **connecté** à sa destination. Ce socket ne pourra communiquer **qu'avec la destination auquel il est désormais connecté**. Si on tentait d'indiquer à ce socket l'adresse IP ou le port de destination par le biais de la fonction `sendto` (voir ci-dessous), les arguments représentant l'adresse / port de destination sont simplement ignorés.

Avec notre socket connecté, on envoie et on reçoit des données via **recv** et **send** :

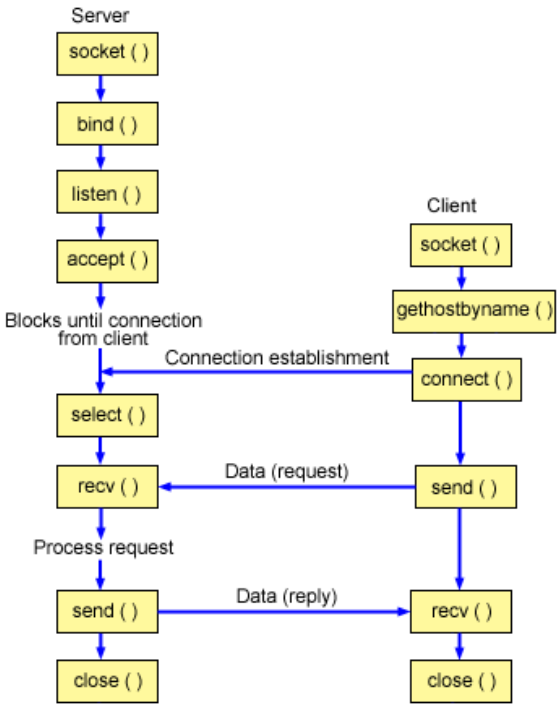
```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Inutile de préciser la destination puisque notre socket est connecté, on précise uniquement le message à envoyer, sa taille, et éventuellement des flags (généralement à 0).

On a implémenté ce mode serveur dans **Durex** (implémentation Durex bind).

La documentation IBM dispose d'une description et d'un exemple de l'implémentation et de l'utilisation de sockets *connection-oriented*, en mode serveur et client :
https://www.ibm.com/docs/en/i/7.3?topic=design-creating-connection-oriented-socket

1 : On a placé les arguments 2 et 3 de **accept** à NULL, mais on pourrait également fournir une structure *sockaddr* (ainsi que sa taille) qui serait remplie par **accept** avec l'adresse IP et le port du client qui se connexte.




### 3. *Connectionless socket usage*

Examinons maintenant l'utilisation et l'implémentation d'un socket SOCK_DGRAM, autrement dit permettant d'établir une connexion UDP avec sa destination.

Dans ce cadre, il n'est pas nécessaire de connecter le socket. Nous ne sommes pas en TCP, la connexion n'est pas bidirectionnelle, on se contente d'envoyer et de recevoir des données sans qu'on ait besoin d'accuser réception etc...

> Pour recevoir des données, il suffit de **bind** le socket à une IP/port puis d'appeler la fonction **recvfrom** :

```
ssize_t recvfrom(int sockfd, void * buf, size_t len, int flags,
                 struct sockaddr * src_addr,
                 socklen_t * addrlen);
```

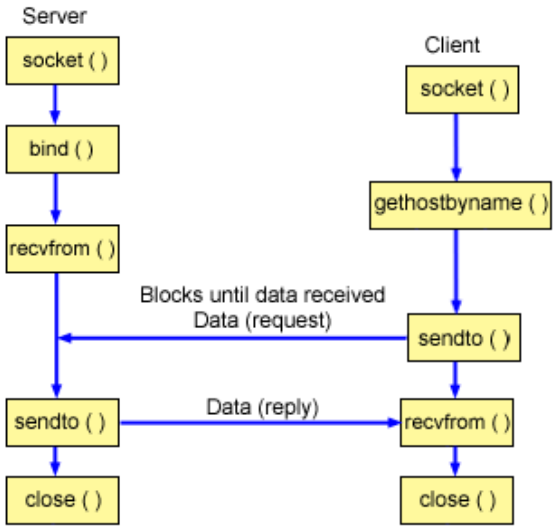
A réception des données, la variable *src_addr* contiendra l'adresse et le port duquel les données ont été reçues.

> Pour envoyer des données, il suffit d'appeler la fonction **sendto** :

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                 const struct sockaddr *dest_addr, socklen_t addrlen);
```

La variable *dest_addr* doit contenir l'adresse et le port où envoyer les données.

Encore une fois, la documentation IBM dispose d'une description et d'un exemple et l'implémentation et de l'utilisation de sockets *connectionless*, en mode serveur et client :




NOTE : On peut appeler **connect** sur un socket SOCK_DGRAM (UDP) afin de définir une adresse de destination par défaut et pouvoir simplement appeler **send()** au lieu de toujours devoir préciser les adresses dans **sendto()**.

4. *Behind the scene : packet handling with TCP and UDP sockets*

Reprenons la structure générale d'un paquet tel qu'il transite par le réseau, avec ses différents headers ajoutés par les différentes couches réseau (voir netwhat, **CCNA_Day2.odt**) :

| [LAYER 2] Data Link Layer | [LAYER 3] Network Layer | [LAYER 4] Transport Layer | |
|---|---|---|---|
| **Ethernet header**<br>- MAC source<br>- MAC dest<br>- Check | **IP header**<br>- IP source<br>- IP dest<br>- Checksum | **TCP/UDP header**<br>- Port source<br>- Port dest<br>- TCP/UDP flags | **DATA** |

Il est intéressant de remarquer que lorsqu'on manipule des sockets SOCK_STREAM ou SOCK_DGRAM, on ne se préoccupe en rien des différents headers ajoutés progressivement par les différentes couches du modèle OSI. Typiquement :

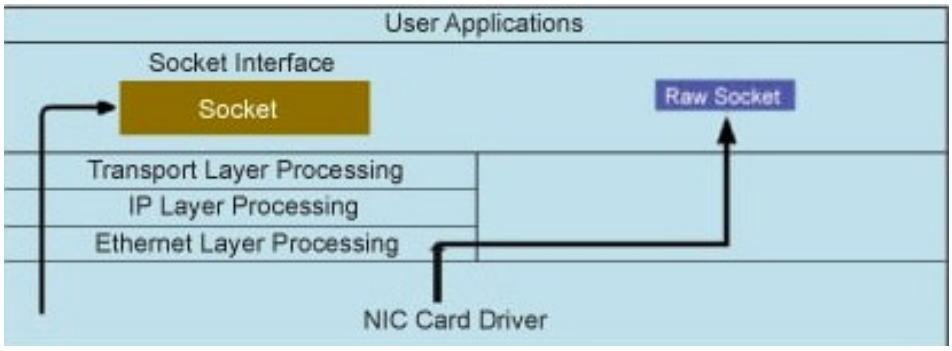
> Quand on appelle recv ou recvfrom, on obtient **uniquement les données**, on ne reçoit pas et donc on ne doit pas parser l'intégralité des headers du paquet.
> Quand on appelle send ou sendto, on indique uniquement à notre programme **les données qu'on souhaite envoyer**, on a pas besoin de construire les headers nécessaires pour que le paquet atteigne sa destination.

C'est précisément l'intérêt de nos sockets en SOCK_STREAM et SOCK_DGRAM. Ces sockets s'occupent du parsing des headers lorsqu'ils reçoivent des données, et de la construction des headers du paquet lorsqu'ils en envoient, pour qu'on ait uniquement à se soucier des données qu'on envoie et qu'on reçoit.

D'ailleurs, il n'est pas possible pour nous de récupérer, à partir de ce type de socket, les informations contenues dans les headers des paquets :

https://stackoverflow.com/questions/47582367/what-headers-are-included-from-a-tcp-stream-when-using-read

Le comportement des sockets classiques est donc représenté par le parcours de gauche sur le schéma suivant :




L'application se charge du traitement des différentes couches précédant la couche applicative pour nous laisser nous concentrer sur les données elles-mêmes.

5. *Raw sockets*

Il existe un autre type de sockets, les **raw sockets**. Ces sockets sont de la famille AF_PACKET :
`raw = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL) );`

Le troisième argument est une macro qui indique qu'on souhaite recevoir sur ce socket l'intégralité des paquets (quel que soit leur type). ETH_P_IP serait par exemple pour ne recevoir que les paquets AF_INET utilisant des adresses IP.

Comme le schéma ci-dessus l'indique, la particularité d'un socket **raw** est qu'il transmet à notre application **le paquet lui-même, sans aucune modification**. Il transmet donc l'intégralité des headers lorsqu'on le reçoit, et lorsqu'on souhaite envoyer des données, il faudra construire ces headers également nous-mêmes.

Notons qu'un socket **raw** peut être associé à un **interface** :
https://man7.org/linux/man-pages/man7/raw.7.html

Il ne pourra cependant jamais être associé à un port spécifique, car le concept de port est un concept TCP/UDP, et qu'on peut tout à fait recevoir des paquets qui ne précisent pas de port (comme ICMP par exemple). Si notre **raw socket** n'est pas associé à une interface (adresse IP locale) spécifique, alors tous les paquets spécifiés à la création de la requête lui sont transmis, de tous les interfaces. Pour limiter notre **raw socket** à un interface particulier, on peut utiliser **setsockopt** :
https://stackoverflow.com/questions/3998569/how-to-bind-raw-socket-to-specific-interface

> Bref, voici la syntaxe nous permettant de recevoir des paquets à partir d'un socket **raw** :

```
unsigned char *buffer = (unsigned char *) malloc(65536);
memset(buffer,0,65536);
struct sockaddr saddr;
int saddr_len = sizeof (saddr);
```

```
//Receive a network packet and copy in to buffer
buflen=recvfrom(sock_r,buffer,65536,0,&saddr,(socklen_t *)&saddr_len);
if(buflen<0)
      return -1;
```

Lorsqu'un packet **raw** est reçu par un appel à **recvfrom**, l'adresse d'origine est parsée malgré tout et transmise dans une structure *sockaddr_ll* :
"*When receiving a packet, the address is still parsed and passed in a standard sockaddr_ll address structure*" ; https://man7.org/linux/man-pages/man7/packet.7.html

D'ailleurs, faisons un petit détour par l'explication de la structure **sockaddr**, sa relation avec **sockaddr_in**, **sockaddr_ll**, **sockaddr_dl** etc... :
https://illumos.org/man/3socket/sockaddr_ll

"The **sockaddr** family of structures are designed to represent network addresses for different networking protocols. The structure struct sockaddr is a generic structure that is used across calls to various socket library routines such as **accept()** and **bind()**. Applications do not use the struct sockaddr directly, but instead **cast** the appropriate networking family specific sockaddr structure to a struct sockaddr *.

Every structure in the sockaddr family begins with a member of the same type, the `sa_family_t`, though the different structures all have different names for the member. For example, the structure struct sockaddr has the following members defined:

```
      sa_family_t     sa_family    /* address family */
      char            sa_data[]    /* socket address (variable-length data) */
```

The member sa_family corresponds to the socket family that's actually in use. The following table describes the mapping between the address family and the corresponding socket structure that's used. The struct sockaddr isn't included since it is a generic structure.

```
            Socket Structure        Address Family
            struct sockaddr_dl      AF_LINK
            struct sockaddr_in      AF_INET
            struct sockaddr_in6     AF_INET6
            struct sockaddr_ll      AF_PACKET
            struct sockaddr_un      AF_UNIX
"
```

Une fois qu'on a reçu le paquet, pour l'exemple du parsing d'un paquet UDP/TCP, voir le tutoriel principal :
https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/

> Pour envoyer des paquets, on utilise sendto. Voici ce qui est fait dans le tutoriel :

```
struct sockaddr_ll sadr_ll;
sadr_ll.sll_ifindex = ifreq_i.ifr_ifindex; // index of interface
sadr_ll.sll_halen = ETH_ALEN; // length of destination mac address
sadr_ll.sll_addr[0] = DESTMAC0;
sadr_ll.sll_addr[1] = DESTMAC1;
sadr_ll.sll_addr[2] = DESTMAC2;
sadr_ll.sll_addr[3] = DESTMAC3;
sadr_ll.sll_addr[4] = DESTMAC4;
```

```
sadr_ll.sll_addr[5] = DESTMAC5;

send_len = sendto(sock_raw,sendbuff,64,0,(struct sockaddr*)&sadr_ll,
sizeof(struct sockaddr_ll));
if(send_len<0)
    return -1;
```

Remarquons qu'on remplit ici le champ sll_addr avec une adresse MAC de destination, ce qui n'est pas nécessaire / ce qui est redondant dans ce cas précis. Lorsqu'on souhaite envoyer des paquets entièrement bruts (AF_PACKET + SOCK_RAW), la structure *sockadd_ll* permet simplement de préciser l'interface qui va devoir s'occuper d'envoyer notre **raw paquet** (identifiée par son index *sll_ifindex*):

"**When transmitting a packet, the user supplied buffer should contain the physical layer header**. That packet is then **queued unmodified to the network driver** of the interface defined by the destination address" ; https://man7.org/linux/man-pages/man7/packet.7.html

Voir également cette réponse stackoverflow :
https://stackoverflow.com/questions/70995951/meaning-and-purpose-of-sockaddr-ll-in-packet-sockets

"Just like `sockaddr_in` identifies an IP address and port, `sockaddr_ll` indentifies a link-layer address and protocol. In the case of raw `AF_PACKET` sockets, the term "address" might be confusing, since we aren't really talking about IP/MAC or whatever other address, but we are talking about an interface (e.g. a physical network device) and a protocol.

An Ethernet header contains metadata about the Ethernet frame, and indicates which protocol (EtherType field) is being encapsulated in the payload. A `sockaddr_ll` can be used to match or to identify a certain interface and protocol (fields `.sll_ifindex` and `.sll_protocol`):

- When a packet socket is bound to a `sockaddr_ll`, it will only receive packets from the specified interface (`.sll_ifindex`) and of the specified protocol (`.sll_protocol`).
- When receiving (**recvfrom** or **recvmsg**), a `sockaddr_ll` structure, which identifies the interface on which the packet was received and the protocol, is populated by the kernel for the user.
- When sending (**sendto** or **sendmsg**), a `sockaddr_ll` structure is used to specify on which interface to send the packet and which protocol to set in the Ethernet header (EtherType field)."

### 6. *AF_INET raw sockets*

https://stackoverflow.com/questions/49309029/confusion-with-af-inet-with-sock-raw-as-the-socket-type-v-s-af-packet-with-so

Jusqu'ici, nous avons étudié des **raw sockets** (SOCK_RAW) définis avec la famille de communication AF_PACKET. Ce qui permet la construction et la manipulation de paquets au plus bas niveau possible, dans le sens où les paquets transmis à l'application ont encore leur *link-level header* (layer 2 avec les adresses MAC), et tous ceux qui viennent après. De même, quand on doit envoyer les données, on doit reconstruire tous les headers jusqu'au header *link-level* de la layer 2[2].

2 : Notons qu'il est possible d'opérer à un niveau un peu plus élevé dans la famille de communication AF_PACKET en

Il est cependant également possible de définir des SOCK_RAW dans la famille de communication **AF_INET**.
https://man7.org/linux/man-pages/man7/raw.7.html
```
s = socket(AF_INET, SOCK_RAW, 0);
```

Lorsqu'un socket de ce type va recevoir un paquet, il traitera **la layer 2** (ethernet header) ainsi que **la layer 3** (IP header), puis transmettra le reste à l'application sans modification (éventuellement une layer 4, transport). Lorsque ce même socket voudra envoyer un paquet, il aujoutera automatiquement les headers de **la layer 3** (adresses IP...) puis les headers de **la layer 2** (adresses MAC...).

En d'autres termes, la différence avec SOCK_STREAM et SOCK_DGRAM de la famille AF_INET est qu'on est libre de parser (en réception) ou de créer (en envoi) la **transport layer** du paquet.

Dans le cadre de ce type de paquet AF_INET/SOCK_RAW, la fonction **sendto** devra avoir en 4ème argument une *sockaddr_in* (castée en *sockaddr*) contenant l'adresse IP de destination afin de pouvoir générer le header IP (le champ relatif au port sera ignoré).

Tableau récapitulatif des headers qui sont générés automatiquement ou laissés à l'utilisateur en fonction des familles et types de socket qu'on vient de voir :

| FAMILY | TYPE | [Layer 2] | [Layer 3] | [Layer 4] |
|---|---|---|---|---|
| AF_PACKET | SOCK_RAW | **X** | **X** | **X** |
| AF_PACKET | SOCK_DGRAM | **O** | **X** | **X** |
| AF_INET | SOCK_RAW | **O** | **~O** | **X** |
| AF_INET | SOCK_DGRAM | **O** | **O** | **O (UDP)** |
| AF_INET | SOCK_STREAM | **O** | **O** | **O (TCP)** |

* AF_INET/SOCK_RAW : si l'option de socket **IP_HDRINCL** est activée, la layer 3 (IP header) n'est pas généré automatiquement et est laissé à l'utilisateur.

* AF_INET/SOCK_RAW : **ATTENTION**, s'il est vrai qu'à l'envoi le header IP est en effet généré automatiquement, **à la réception** le header IP n'est pas parsé automatiquement et est **transmis avec le paquet** :
"*The IPv4 layer generates an IP header when sending [...]. For receiving, the IP header is always included in the packet*" ; man7 raw.

---

créant un socket SOCK_DGRAM. Dans le cadre de cette famille de communication, SOCK_DGRAM permet la construction automatique du *link-level header* de la layer 2 (avec les adresses MAC etc...) à l'envoi de donnée, et son parsing automatique à la réception.

## II. PROTOCOLE ICMP ET PAQUETS ICMP

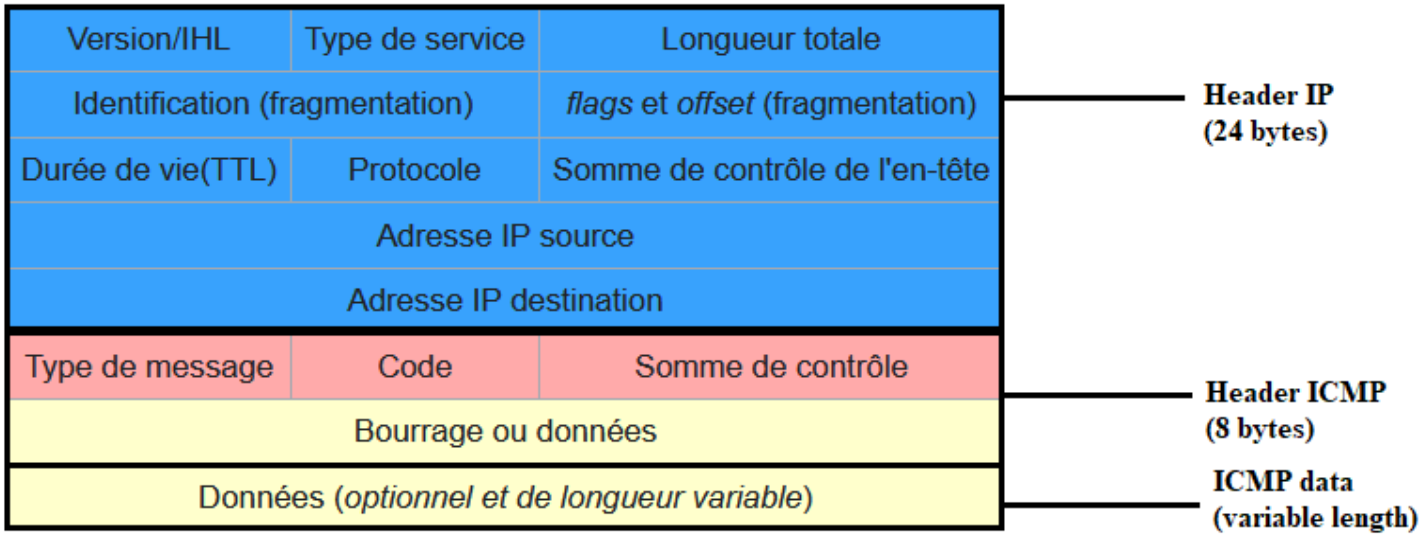
De la page Wikipedia de ICMP :
"*ICMP is part of the Internet protocol suite as defined in RFC 792. ICMP messages are typically used for diagnostic or control purposes or generated in response to errors in IP operations. ICMP errors are directed to the source IP address of the originating packet.*

*For example, every device (such as an intermediate router) forwarding an IP datagram first decrements the time to live (TTL) field in the IP header by one. If the resulting TTL is 0, the packet is discarded and an ICMP time exceeded in transit message is sent to the datagram's source address*".

"*ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, but to send error messages and operational information*".

Les paquets transportant des messages ICMP utilisent donc le protocole IP afin d'atteindre leur machine de destination. Les paquets ICMP n'ont **aucune transport layer** (contrairement à TCP et UDP), autrement dit ils présentent les headers **ethernet** (layer 2, *data-link*), puis **IP** (layer 3, *network*), et ensuite directement viennent les données ICMP. C'est pourquoi ICMP est considéré comme un *network* protocole (relevant de la layer 3).

Voici la structure des données ICMP (on intègre le header IP) :




- En bleu, un **header IP** avec *Type de service* (service type) valant 0, et *Protocole* (transport) valant 1.
- Un type de message ICMP (1 byte).
- Un code identifiant l'erreur (1 byte).
- Un **checksum** (2 bytes) calculée sur la partie spécifique à ICMP (sans l'en-tête IP).
- D'une partie aménagée pour des données relatives aux différents types de réponses (4 bytes). Si elle n'est pas utilisée, on la remplit. Cette partie peut correspondre aux Identifiant et Numéro de séquence pour un paquet de type Ping par exemple.
- Du message, de taille variable.

La liste des **types et codes** de messages ICMP peut être trouvée ici :

https://fr.wikipedia.org/wiki/Internet_Control_Message_Protocol

Pour notre commande ping, nous intéressent les **types 0 et 8** (qui n'ont pas de *code*, leur code sera toujours 0) :
> Type 8, **Echo Request**.
> Type 0, **Echo Reply**.

Ce qu'on fait, lorsqu'on souhaite *ping* une machine distante, c'est simplement de lui envoyer un message ICMP de type 8 (*echo request*), et d'attendre en retour un message ICMP de type 0 (*echo request*). Il est possible qu'au lieu de recevoir en réponse ICMP type 0, on reçoive :

- Un message ICMP qui nous indique une erreur dans la transmission de notre message originel, comme ICMP type 3 **Destination Host Unreachable**, retourné par notre routeur qui ne trouve pas de route appropriée pour transmettre le paquet (par exemple, une adresse IP qui se trouve dans notre subnet mais qui ne correspond pas à une machine : le paquet est envoyé au routeur, qui nous renvoie un ICMP type 3 car il ne peut pas transmettre le paquet à la machine désirée).
- Rien du tout : le message ICMP est parti, mais n'atteint jamais la machine cible qui ne nous transmet donc aucune réponse (par exemple, une adresse IP privée qui ne se situe pas dans notre subnet, le routeur va tenter de transmettre le paquet en agissant comme un *gateway*, mais le paquet n'atteindra jamais sa destination).

## III. FT_PING CODE

Base :
https://www.geeksforgeeks.org/ping-in-c/

### 1. *DNS resolution*

La première étape, avant de faire quoi que ce soit d'autre, est de récupérer **l'adresse IP de destination** pour notre commande **ping**. En effet, le premier argument du programme peut certes être une adresse IP, et dans ce cas on la connaît, mais il peut également être un nom de domaine (comme **google.com**), et dans ce cas il faut opérer une résolution DNS afin de connaître l'adresse IP cible à laquelle on va envoyer nos paquets ICMP.

La fonction qui est en charge de la résolution DNS est la fonction **dns_lookup.c** dans le fichier **dns.c**. On se sert de `getaddrinfo` (au lieu de `gethostbyname` comme dans le tutoriel, car il est indiqué que c'est maintenant préféré :
https://man7.org/linux/man-pages/man3/getaddrinfo.3.html

Comme indiqué dans le man, voici le prototype de la fonction :

```
int getaddrinfo(const char *restrict node,
                      const char *restrict service,
                      const struct addrinfo *restrict hints,
                      struct addrinfo **restrict res);
```

Et voici à quoi correspondent les structures **addrinfo** qui sont manipulées :

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

La structure **hints** permet de spécifier à la fonction de résolution DNS un certain nombre de filtres :

- `ai_family` : desired address family for returned adresses in ai_addr (IPv4 or IPv6 or both), 0 is any.
- `ai_socktype` : desired socket type for results in ai_addr (SOCK_STREAM, SOCK_DGRAM...), 0 is any.
- `ai_protocol` : desired protocol, 0 is any.

L'argument **node** correspond au nom de domaine qu'on souhaite résoudre. L'argument **service** permet de configurer le port pour chaque socket retourné par la fonction.

L'argument **res** est un pointeur (initialisé à 0), que la fonction **getaddrinfo** fera pointer sur une liste chaînée de structures **addrinfo** qu'elle aura alloué dynamiquement et qui contiendront les résultats de la résolution DNS (**les** résultats car un seul nom de domaine peut être *multihomed*, accessible via IPv4 et IPv6 etc...).

```
int dns_lookup(t_ping *p)
{
    struct addrinfo         hints;
    int                     s;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;

    s = getaddrinfo(p->target, NULL, &hints, &(p->lookup_output));
    if (s != 0)
    {
        fprintf(stderr, "ping: %s: %s\n", p->target, gai_strerror(s));
        return (1);
    }
    p->addr_con = (struct sockaddr_in *)(p->lookup_output->ai_addr);

    return (0);
}
```

On voit que la fonction est assez simple, on crée une structure **addrinfo** qui contiendra nos *hints*, et on initialise tous ses membres à 0, sauf **ai_family** : on ne souhaite en résultat que des adresses IPv4. On appelle ensuite **getaddrinfo** avec en premier argument notre cible (argv[1] en réalité), aucun port de spécifié, et l'adresse d'un pointeur sur une structure **addrinfo** qui contiendra nos résultats.

On stock ensuite (si tout s'est bien passé) notre résultat (en réalité le premier élément de la liste chaînée de résultats, qui devrait être le seul ici) dans une structure **sockaddr_in** : ce qui nous intéresse surtout est que cette dernière contiendra l'adresse IP associée au nom de domaine dans

`addr_con->sin_addr`.

Remarquons que si le client nous a fourni une adresse IP (par exemple 127.0.0.1), la fonction **getaddrinfo** ne va rien résoudre et simplement placer l'adresse IP telle quelle dans son résultat (`lookup_output->ai_addr`).
Pour free la structure allouée par **getaddrinfo**, il nous suffit d'appeler **freeaddrinfo**, voir man7.

### 2. *Reverse DNS resolution*

Comme on le sait, une résolution DNS inverse permet d'associer un nom de domaine à une adresse IP. On effectue dans notre programme également un *reverse DNS lookup*, même s'il s'agit surtout de raisons esthétiques : si l'utilisateur nous a fourni une adresse IP, on effectue le *reverse lookup* pour lui indiquer le nom de domaine correspondant.

Il est tout à fait possible que le *reverse DNS lookup* ne réussisse pas, ce qui n'empêche pas le programme de suivre son cours, ce qui importe étant l'adresse IP de destination qu'on a récupéré avec le lookup DNS simple.

On utilise la fonction **getnameinfo**, voir man :
https://man7.org/linux/man-pages/man3/getnameinfo.3.html

### 3. *Socket*

Une fois qu'on dispose de l'adresse IP de destination, il nous faut également ouvrir un **socket** qui devra envoyer nos paquets ICMP. On va pour cela choisir un socket **AF_INET/SOCK_RAW**, en précisant le protocole **IPPROTO_ICMP** :

```
p.sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

> AF_INET/SOCK_RAW, comme indiqué ci-dessus, va nous permettre d'envoyer des paquets avec un header IP automatiquement généré par le kernel, mais sans couche **transport**, car les paquets ICMP n'en ont pas.

> IPPROTO_ICMP permettra de ne recevoir, sur ce socket, **que** les paquets ICMP récupérés par nos interfaces réseau.

On va également utiliser **setsockopt** pour configurer notre socket, de deux manières. Premièrement, pour la configuration du **Time To Live** qui se situe dans le header IP (d'où SOL_IP et IP_TTL) :

```
p->ttl = 64
[... SNIP ...]
setsockopt(p->sockfd, SOL_IP, IP_TTL, &(p->ttl), sizeof(p->ttl));
```

Le **Time To Live** est une valeur, dans le header IP, qui indique combien de *hops* maximum un paquet peut circuler sur le réseau. Par exemple, si le **Time To Live** est de 64, il pourra passer par 64 routeurs maximum : chaque routeur va retirer 1 à cette valeur du TTL ; si elle tombe à **0**, alors le paquet est *drop* par le routeur, qui renvoie à l'émetteur un paquet ICMP type 11 (**Time exceeded**).

Deuxièmement, pour la configuration du **timeout** pour nos appels à **recvfrom** (le temps qu'on va

attendre pour qu'une machine à qui on a envoyé un **ICMP Echo request** nous réponde) :

```
setsockopt(p->sockfd, SOL_SOCKET, SO_RCVTIMEO, (const char *)&(p->recv_timeout), sizeof(p->recv_timeout));
```

4. *Construire le paquet ICMP et l'envoyer*

Notre socket est prête, construisons maintenant le paquet **ICMP Echo request** qu'on va envoyer à notre machine de destination. Rappelons qu'on paquet ICMP est constitué :

- D'un header ICMP (8 bytes).
- De données (longueur variable).

Pour le header du paquet, on va utiliser la structure **icmphdr** définie dans **<netinet/ip_icmp.h>** :

```
struct icmphdr
{
  u_int8_t type;                /* message type */
  u_int8_t code;                /* type sub-code */
  u_int16_t checksum;
  union
  {
    struct
    {
      u_int16_t        id;
      u_int16_t        sequence;
    } echo;
    u_int32_t        gateway;
    struct
    {
      u_int16_t        __unused;
      u_int16_t        mtu;
    } frag;
  } un;
};
```

Les seuls membres de cette structure qui vont nous intéresser :

- **Type** : pour le type de message ICMP.
- **Code** : pour le code associé.
- **Checksum** : pour y insérer le checksum de vérification.
- (**id** : éventuellement pour identifier d'où viennent les paquets ICMP).
- (**sequence** : éventuellement pour indiquer combien de paquets on a envoyé jusqu'ici dans l'exécution de la commande).

Pour les données, un simple tableau de char qu'on remplira de manière aléatoire fera l'affaire. On va faire en sortes que notre paquet ICMP fasse en tout 64 bytes[3], on va donc définir 56 bytes pour ce tableau représentant les données, comme ça on arrivera à 64 avec les 8 bytes du header.

Voici donc la structure qui représentera notre paquet ICMP :

3 : Il n'est pas obligatoire que le paquet ICMP transportant une **ICMP Echo request** fasse 64 bytes, on peut lui donner une taille arbitraire en ajustant la taille du tableau contenant les "données".

```
/* Structure used to send ICMP Echo request */
typedef struct		s_pingpckt
{
	struct icmphdr	hdr;
	char		msg[PING_PACKET_SIZE - sizeof(struct icmphdr)];
}			t_pingpckt;
```

Et voici comment on va l'initialiser avant d'envoyer notre paquet :

```
void build_ping_pckt(t_ping *p)
{
    size_t i;
    bzero(&(p->pingpckt), sizeof(p->pingpckt));

    p->pingpckt.hdr.type = ICMP_ECHO;
    p->pingpckt.hdr.un.echo.id = getpid();

    for (i = 0; i < sizeof(p->pingpckt.msg) - 1; i++)
        p->pingpckt.msg[i] = i + '0';
    p->pingpckt.msg[i] = '\0';

    p->pingpckt.hdr.un.echo.sequence = p->msg_sent_count++;
    p->pingpckt.hdr.checksum = checksum(&(p->pingpckt), sizeof(p->pingpckt));
}
```

De manière assez simple, on définit le type de paquet ICMP comme **8** (**ICMP Echo request**), on donne un **id** (pas nécessaire), on remplit aléatoirement le tableau contenant les données, on indique le numéro du message (pas nécessaire), et on calcule le checksum (voir fonction dans le même fichier pour le calcule du checksum).

Suite à l'appel à notre fonction **build_ping_pckt**, dans la fonction **ping_loop**, on appelle donc **sendto** afin d'envoyer notre paquet à notre adresse de destination, qu'on a stocké lors de la résolution DNS dans `p->addr_con` :

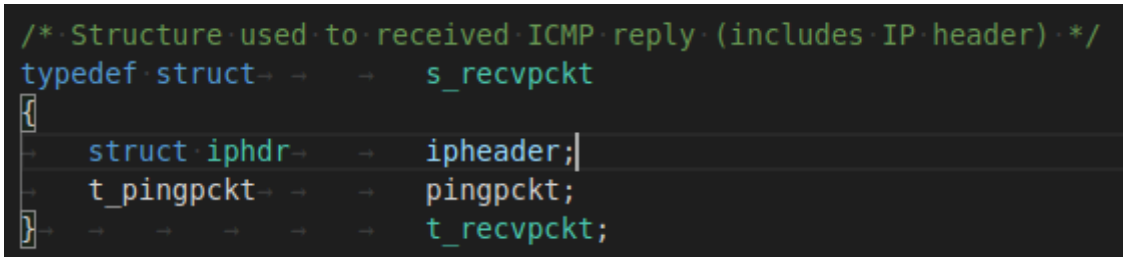
```
sendto(p->sockfd, &(p->pingpckt), sizeof(p->pingpckt), 0, (struct sockaddr *)p->addr_con, sizeof(*(p->addr_con));
```

5. *Réception du paquet ICMP Echo reply*

Une fois notre *ICMP Echo request* envoyée, on appelle **recvfrom** afin d'écouter la réponse **ICMP** de la machine distante, qui devrait, si tout se passe bien, nous répondre :

```
recvfrom(p->sockfd, &(p->recvpckt), sizeof(p->recvpckt), 0, (struct sockaddr *)r_addr, &addrlen);
```

Remarquons qu'à la réception de la réponse de la machine distante, on place les données que cette dernière nous a envoyé dans une structure **recvpckt**, et non **pingpckt** (la structure qui représentait notre paquet ICMP). Cela est dû au fait qu'à la **réception** de paquets d'une socket AF_INET/SOCK_RAW, le **header IP est inclu** dans ce qu'on reçoit. Ainsi, voici la structure **recvpckt** :




On voit qu'il se compose à la fois du header IP **qu'on reçoit d'abord** (c'est pourquoi il est en haut de la structure, les données reçues le rempliront d'abord), puis du paquet ICMP Echo reply qui vient ensuite.

Une fois un tel paquet de réponse reçu, on parse et on s'occupe du paquet ICMP qui nous a été renvoyé en accédant à **pingpckt** :

```
if (!(p->recvpckt.pingpckt.hdr.type == 0 && p->recvpckt.pingpckt.hdr.code
== 0))
{
      [We received an ICMP response indicating an error...]
}
```

6. *Divers*

Le reste du programme est relatif à des détails d'implémentation de ping, de calcule du temps passé, et d'affichage des erreurs. Pas vraiment de concept nouveau qu'il faille explorer dans cette documentation.

Une remarque peut-être: dans le tutoriel de **geeksforgeeks**, une erreur est présente dans le code puisque quand ils reçoivent la réponse d'une machine distante à leur demande de ping (ICMP Echo request), ils ont oublié que dans les données reçues se trouvent **d'abord le header IP** ; ils ne le parsent pas. Ils considèrent donc que le **type** indiquant une ICMP Echo reply est **69** au lieu de **0**, alors que **69** correspond simplement aux premiers bytes du **header IP** (la version de IP utilisée). Ce qui fait qu'ils ne verront pas les erreurs ou les vrais types / codes / données du paquet ICMP qu'ils reçoivent, puisqu'ils examinent en réalité les données du header IP...
http://jptosoni.over-blog.com/2021/03/icmp-type-69-code-192.html

Pour mesurer le temps, **gettimeofday** est **obsolète** (norme POSIX 2008) :
https://stackoverflow.com/questions/12392278/measure-time-in-linux-time-vs-clock-vs-getrusage-vs-clock-gettime-vs-gettimeof

Utiliser **clock_gettime** à la place.