Projet réseau et système 31 mai 2008

Rapport de projet

IP over UDP

Encadrant : ALLALI Julien

BEN LAKHAL Marouane
EL AFRIT Mohamed Amine
HABASSI Seif Eddine
KHABOU Amal
LAVERGNE Julien

Table des matières

Table des figures

1

Présentation et comprhension du sujet

1.1 Présentation du sujet

Le projet Ip_over_UDP consiste en l'implantation d'un réseau IP virtuel et des services associés sur le support UDP réel. La transmission des packets sur le réseau se fera par broadcast UDP.

Il nous est donc demandé de mettre en place et de pouvoir manipuler un réseau virtuel et ceci en implantant les commandes suivantes :

- **interface** :
  - ajout d'une interface.
  - suppression d'une interface.
  - affichage de l'ensemble des interfaces réseau.

- **ifconfig** :
  - activation d'une interface déjà créée.
  - désactivation d'une interface déjà créée.
  - configuration d'une interface déjà créée.

- **route** : ajout, suppression, activation, désactivation, affichage des entrées de la table de routage.

- **arp** : affichage du cache ARP.

- **ping** : envoie des paquets ICMP à l'adresse IP spécifiée

Le réseau réalisé doit être conforme aux spécifications suivantes :

- RFC 791 pour IP.
- IEEE 802.3 pour Ethernet.
- RFC 826 pour ARP.
- RFC 792 pour ICMP.

1.2 Compréhension du sujet

Une machine du réseau virtuel correspond à une instance du programme principal. Sur une machine réelle, il est possible de lancer plusieurs instances du programme.

L'ensemble de ces machines virtuelles constitue le réseau virtuel à réaliser où les ports servent de filtres pour la diffusion de packets envoyés en broadcast sur le réseau réel. En effet, un domaine de diffusion est

composé de toutes les machines qui écoutent sur un port UDP donné. Lorsque sur une machine, le programme utilise plusieurs ports UDP, cette machine joue le rôle de passerelle entre les différents réseaux locaux.

2

Découpage en module

Nous avons découpé le code de notre application en modules comme suit :

- Le module *ethernet* `ether_header.h` : il contient la structure de l'entête ethernet et l'*API* pour sa manipulation.

- Le module *ip* `ip_header.h` : il contient la structure de l'entête ip et l'*API* pour sa manipulation.

- Le module *icmp* `icmp.h` : qui contient la structure de l'entête icmp et l'*API* pour sa manipulation.

- Le module *arp* `arp.h` et `arp_header.h` : qui contient la structure de l'entête arp, l'*API* pour sa manipulation, les fonctions d'envoi et de la réception des requêtes *ARP*.

- Le module *interface* `interface.h` : qui permet la manipulation des interfaces.

- Le module *config* `ifconfig.h` : qui permet la configuration des interfaces, des adresses IP, ...,

- Le module *routage* `route.h` : qui permet la configuration de la table de routage.

- Le module *test* `src/test/*` : pour tester le bon remplissages des champs des structures.

- le module *ping* `ping.h` : pour envoyer des *ping*,

3

Les diffrents enttes

3.1 en-tête ETHERNET

L'entête ethernet a la structure suivante :

```
typedef struct _ether_header
{
  uint8_t dst[6];
  uint8_t src[6];
  uint16_t type;
} ether_header;
```

voir figure 3.1

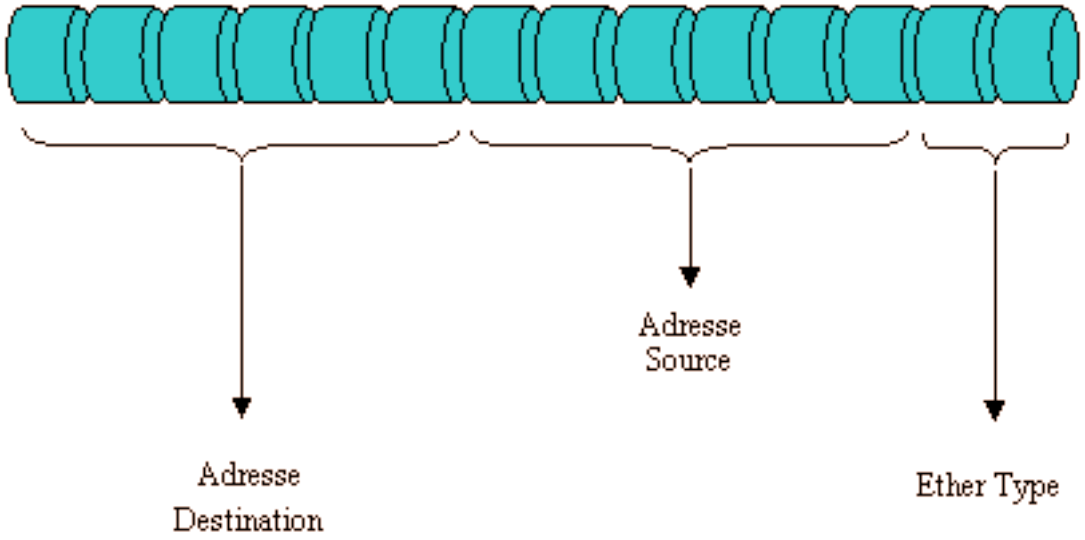

Fig. 3.1 – Entête ethernet - 14 octets

3.2 en-tête IP

L'entête ip a la structure suivante :

```
typedef struct _ip_header {

  uint8_t  version;
  uint8_t  IHL;
  uint8_t  TOS;
  uint16_t total_length;
  uint16_t id;
  uint8_t  flags;
  uint16_t offset;
  uint8_t  ttl;
  uint8_t  protocol;
  uint16_t checksum;
  uint32_t src;
  uint32_t dst;

} ip_header;
```

voir figure 3.2

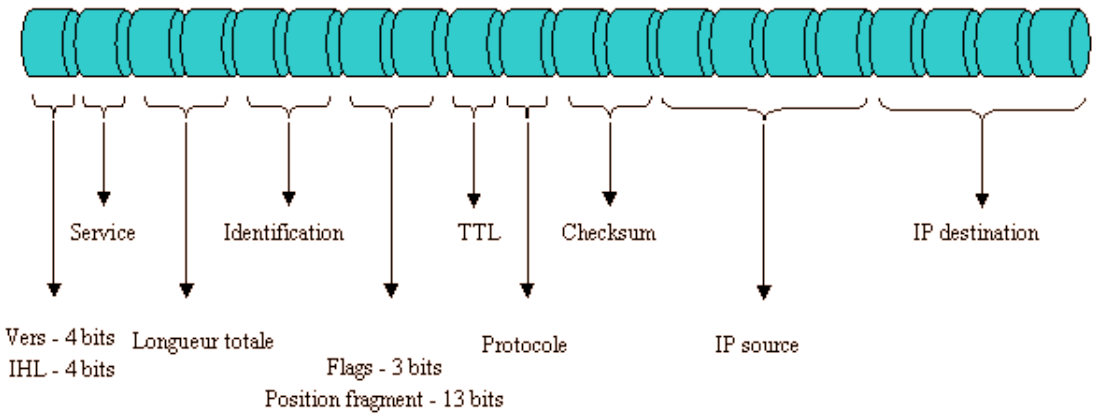

Fig. 3.2 – Entête ip - 20 octets

3.3 en-tête ARP

L'entte ARP a la structure suivante :

```
typedef struct arp_header_{
  uint16_t hardware_type;
  uint16_t protocol_type;
  uint8_t hardware_length;
  uint8_t protocol_length;
  uint16_t operation;
  uint8_t sender_hardware_address[6];
  uint32_t sender_protocol_address;
  uint8_t target_hardware_address[6];
  uint32_t target_protocol_address;
} arp_header;
```

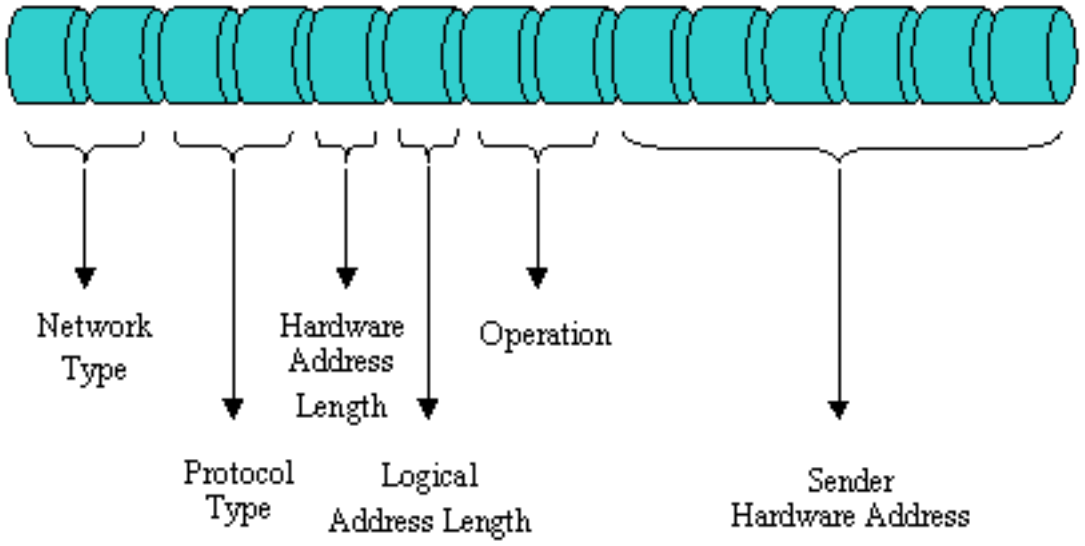

Fig. 3.3 – Premire partie de l'entte arp

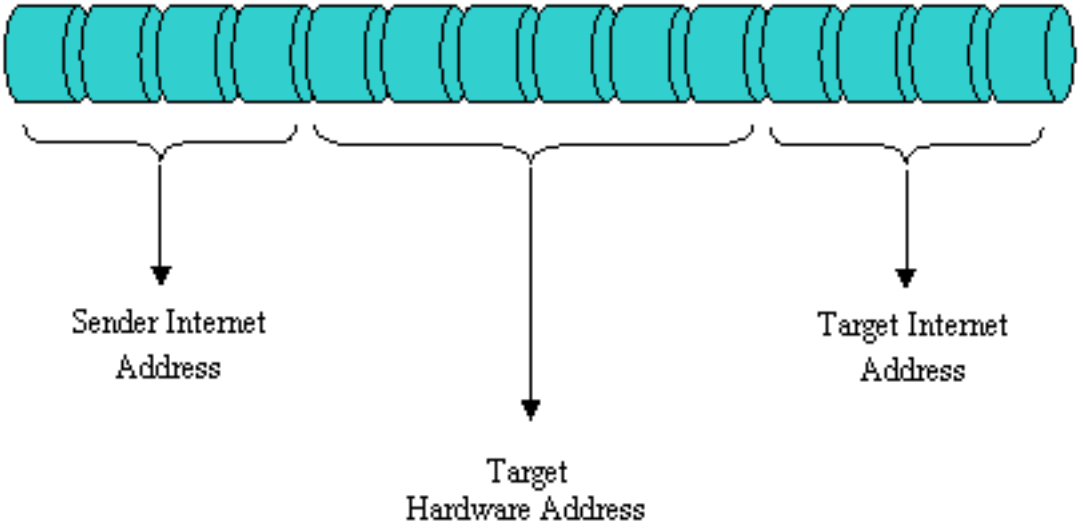

Fig. 3.4 – Deuxime partie de l'entte arp

3.4 en-tête ICMP

L'entête icmp a la structure suivante :

```
typedef struct _icmp_header
{
```

```
  uint8_t  type;
  uint8_t  code;
  uint16_t checksum;
  uint16_t id;
  uint16_t seq_number;

} icmp_header;
```

voir figure 3.5

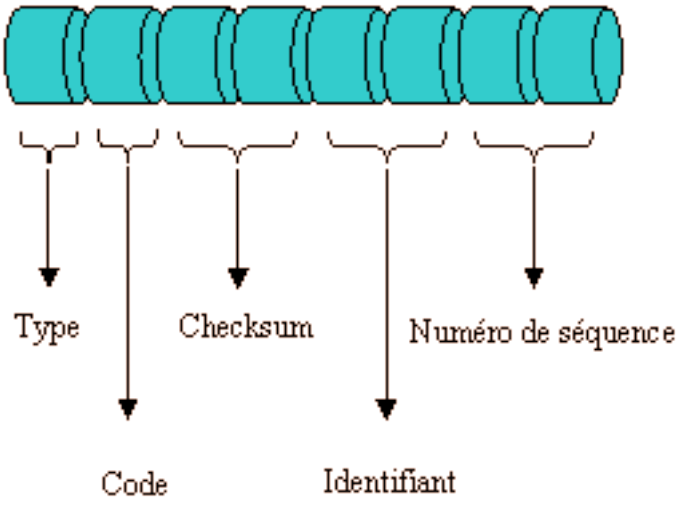

Fig. 3.5 – Entête icmp - 8 octets

4

Quelques points importants

4.1 Interfaces

Dans ce projet, il nous a été demandé de créer un "espace" de travail virtuel. Par exemple, un sous-réseau peut être simulé par un ensemble d'interfaces qui sont connectées sur un même port.
En effet, notre programme offre à l'utilisateur la possibilié de créer jusqu'à 250 interfaces sur sa machine. Chaque interface est considérée comme une machine ayant une addresse IP, une adresse MAC... On présente alors dans cette partie notre structure de l'interface et comment se fait l'ajout et la suppression d'une interface dans la liste chainée.

4.1.1 La structure interface

Comme expliqué précédemment, une interface joue le rôle d'une machine, et donc tout comme une vraie machine, elle doit posséder un nom, une adresse IP, une adresse MAC, un port qui lui est associé et même un "processeur" afin de gérer quelques tâches.
Pour ce faire, on a défini la structure interface_element comme suit :

```
typedef struct _interface_element
{
  uint8_t id;
  uint16_t port;
  uint32_t ip;
  uint8_t mac[6];
  int mtu;
  bool is_active;   //pour verifier si l'interface est active (1) ou pas(0)
  pthread_t thread; //ceci va representer un processus qui tourne en permanene
                    // et traiter les differentes commandes qui l'interesse
  struct _interface_element* next;
} interface_element;
```

Les cinq premiers champs contiennent les différentes informations caractérisant une interface.

- Le champ **is_active** indique l'état de notre interface, à savoir, si elle est active ou pas : Dans le cas où elle ne l'ai pas une interface ne fait rien, elle n'accepte aucun message et n'en envoi aucun.
- Le champ **thread** est le champ qui joue le rôle du "processeur", en effet, celui-ci va pointer sur le thread créé qui gére cette interface.
- Le champ **next** est utilisé pour parcourir notre liste d'interfaces.

4.1.2 La liste des interfaces

Au début, on stockait nos interfaces dans un tableau statique, mais pour éviter le gaspillage de memoire, on a opté pour les listes chainées.
Cette liste est manipulée à l'aide d'un pointeur de structure qui donne l'emplacement de la dernîre interface ajoutée.

4.1.3 L'ajout et la suppression des interfaces

Lors de l'ajout d'une interface, on doit faire attention à ne pas donner un nom déjà existant (car chaque interface est identifée par son nom). Dans le but de faciliter cete opération, on a présenté les noms par des entiers. De plus, si une interface est supprimée son nom peut être attribué à une nouvelle interface.

4.2 Trame ethernet

Une trame ethernet est constitué des différents entêtes encapsulées aux données qu'on souhaite envoyer.

On commence par encapsuler les entêtes et on envoie un *ping request* ou bien un *arp request* suivant le résultat de la recherche de l'adresse MAC du destinataire dans l'ARP cache de la source.

4.3 Le routage

Le routage se fait au moyen d'une table de routage. Celle-ci est implantée par une liste chaînée simple déclarée en variable globale dans le programme. La structure d'un élément route est la suivante :

```
typedef struct _route_element
{
  uint32_t dst;
  uint32_t netmask;
  uint32_t gw;
  int id;
  bool is_active;
  struct _route_element* next;
} route_element;
```

Pour des raisons de simplicité, on a choisi de prendre un entier pour l'option *dev* au lieu d'une chaîne de caractères, conctretèment on a :

```
1  route add -host w.x.y.z dev 0
```

au lieu de :

```
1  route add -host w.x.y.z dev eth0
```

Si on n'utilise pas l'option dev on utilise forcément l'option *gw* qui permet de préciser la passerelle au quel cas on asssocie l'interface à l'hôte en parcourant la table de routage pour récupérer le numéro d'interface correspondant à la passerelle donnée.
Les éléments *route* sont ajoutés à la table de telle sorte que celle-ci reste trieée en fonction du *netmask*. Cela permet d'avoir une route minimale lors du routage.

4.4 ARP

Pour envoyer un datagramme *IP*, la machine source doit connaître l'adresse *MAC* du destinataire. Il était donc nécessaire de mettre en place un module *ARP* permettant la résolution des adresses *MAC* dont on a besoin pour l'envoi des datagrammes *IP*. En effet, dans le cas où l'adresse *MAC* du destinataire n'est pas connue, des *ARP requests* sont envoyées aux différentes machines du réseau.

La machine concernée par l'*ARP request* renvoie un *ARP reply* la machine source qui ajoute ce résultat à son *ARP cache*. En effet, le *cache ARP* est une table contenant l'adresse *IP* et l'adresse *MAC*. On a implanté cette table par une liste chaînée. Les entrées les plus récentes sont ajoutées en tête de la liste respectant ainsi l'intérêt du cache. Il a fallu limiter la taille du *cache ARP* en ajoutant un compteur du nombre des entrées qui ne doit pas dpasser le *ARP_CACHE_SIZE*. Dans ce cas, une fois le cache est rempli, l'entrée la plus ancienne est supprimée et la plus récente est ajoutée en tête de la liste.

Lors du routage d'un *datagramme IP*, une première étape consiste en le parcours du *ARP cache*, si aucune entrée ne correspond à l'adresse ip du destinataire, un *ARP request* est envoyé. Ces deux étapes s'enchainent tantqu'on n'a pas trouvé le résultat recherché et qu'on n'a pas dépass un cetain temps *ARP_TIMEOUT* qu'on a fixé à 10.

4.5 Envoi

Les fonctions d'envoi sont implantées dans le fichier *ping.c*. Elle apelle une fonction gnrale d'envoi qui a t utilise pour les pings et les requtes ARP. Pour le ping, les primitives d'envoi sont les suivantes :

```
void send_ping(interface_element*, ether_header*, ip_header*, icmp_header*, int);
void send_ping_request(uint32_t, uint8_t, uint8_t, int, int);
void send_ping_getway(uint32_t, ether_header*, ip_header*, icmp_header*, int);
```

L'envoi est prcd de la construction des paquets. Suivant la nature du message envoyer : ping ou requte ARP, on ajoute les enttes ncessaires. Ainsi, lorsqu'on fait un *ping* sur une machine, on envoie une requête *icmp* à la machine destination. Une fois les différents entête construits, ils sont encapsulés dans un paquet. L'exemple suivant illustre cette encapsulation :

```
set_ether_data(etherh, data);
set_ip_data(iph, data);
set_icmp_data(icmph, data);
```

Dans cet exemple *data* correspond au paquet à envoyer, *etherh* correspond à l'entête ethernet, *iph* correspond à l'entête ip et *icmph* correspond à l'entête icmp.

4.6 Réception

Les fonctions de réception ont été implantées dans le fichier *ping.c* pour les pings et dans le fichier *arp.c* pour les requêtes ARP. Les primitives de reception pour le cas du ping sont les suivantes :

```
void* receive_ping_request(void* pta);
void* receive_ping_reply(void* pta);
void* receive_ping_getway(void* pta);
```

A la réception d'un message on opère la décapsulation des différents entêtes, puis la réalisation du traitement approprié suivant le cas. A l'inverse de la fonction *set_ether_data*, la fonction *get_ether_data* permet de récuperer l'entête ethernet à partir du paquet reçu. De la même façon les autres entêtes sont decapsulés par des fonctions similaires : *get_ip_data()* et *get_icmp_data()* pour le cas du ping et *get_arp_data()* pour le cas d'une demande ARP.

4.7 Programmation des threads

Lors de ce projets, les pocessus légers ont joué un rôle important. En effet, vu que les interfaces sont considérées comme des machines, il a été intêressant de travailler avec les threads.

On a préféré travailler avec les threads plutôt qu'avec des appels systèmes *fork()* car ceux ci ont l'avantage de partager le même espace memoire et assure alors le bon transfert des paramêtre.

Ces processus légers sont créés quand il s'agit de lancer un processus manipulant les données d'une interface. Par exemple, en activant une interface avec la commande *ifconfig interface up* un thread se met en travail et reste en écoute sur un port afin de recevoir les donnéees envoyées par les autres interfaces.

Si cet interface recoit un ping ou une demande arp, le thread asocié crée un nouveau thread qui s'occupe de l'envoie de réponse alors que le premier reste en écoute.
Une fois l'interface est désactivée ce thread se met en attente.

On a aussi géré les accés concurrents à l'aide de MUTEX qui verrou une variable par un thread si celle-ci va être modifiée.

5 Organisation du groupe

5.1 Gestion du code

Nous étions cinq à travailler sur le projet. Pour la gestion du code source nous avons utiliser le dépôt **subversion** proposé sur la forge de l'ENSEIRB, ce qui nous a permis, suite à la décomposition des tâches, de travailler d'une façon indépendante tout en gardant une vision générale sur l'avancement global du projet et sur l'avancement de chaque membre de l'équipe.
Concernant le codage, aprés une longue phase de compréhension du sujet, qui d'ailleurs explique l'absence d'un traitement complet de certains points du projet tels que la fragmentation ou le TTL, nous nous sommes mis d'accord sur l'architecture globale du projet ainsi que les modules à réaliser, en se fixant des conventions de codage et en décidant de documenter d'une façon concise ce qui n'est pas explicite dans le code. Le développement des différents modules a été dirigé par les tests et le contrôle des appels système.

5.2 Communication interne

Nous avons organisé des réunions régulièrement surtout au démarrage du projet et ceci pour analyser et comprendre le sujet. Une fois les tâches sont réparties, nous avons communiqué par mails pour se poser des questions ou pour se fixer de nouvelles réunions. Nous avons aussi profité du système de log de svn pour décrire ce qui est fait et ce qui reste à faire.

5.3 Répartition des tâches

La décomposition de l'application en modules nous a permis de se répartir les tâches. Chacun a principalement travaillé sur une des parties du projet, tout en gardant une communication régulière avec les autres membres Nous avons essayé de mettre le plus souvent le travail de chacun en commun afin de faciliter l'intégration des différentes parties.

6 Conclusion

Lors de la ralisation de ce projet, nous avons pu simuler un rseau de communication entre diffrentes machines. Ce projet nous a permis d'approfondir nos connaissances sur le fonctionnement des rseaux IP et de mieux comprendre les mcanismes de routage, de cache ARP et de fragmentation. Il a galement t l'occasion de mettre en oeuvre le manipulation des threads et la gestion des accs concurrents aux donnes bienque nous nous sommes plutt focaliss sur la partie rseau du projet.

Plusieurs fonctionalités ont été implémentées et testées, à savoir : la configuration des interfaces, des tables de routage, de manipulation des différents entêtes et données envoyées, de l'ARP, ...

Cependant quelques points assez importants tels que la fragmentation et la gestion du TTL n'taient traits que partiellement.