

# VII - Communication UDP

# Communication UDP

## UDP versus TCP

- Dans la communication par flux (comme **TCP**)
  - Les informations sont reçues par l'application **dans l'ordre** de leur émission, **intègres** et **sans perte**
  - nécessité d'initier une connexion par un handshake  $\Rightarrow$  ressources dédiées à cela, délais
  - toute donnée reçue doit être vérifiée et acquitée  $\Rightarrow$  ralentit le débit
- Dans la communication par paquet (comme **UDP**)
  - il n'y a **pas d'ordre** dans la délivrance des paquets à l'application réceptrice. Un paquet posté en premier peut arriver en dernier
  - il n'y a **pas d'intégrité** des données reçues garantie.
    - $\rightarrow$  si l'entête du paquet UDP contient un champs checksum non nul, alors l'intégrité des données est vérifiée.
    - $\rightarrow$  en IPv6, le champs checksum doit être non nul sauf pour les protocoles tunnel.
  - communication **non fiable** : un paquet envoyé peut être perdu

# Communication par paquets

## UDP

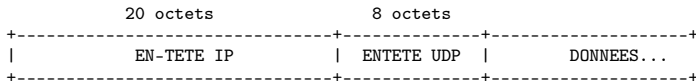
UDP est donc un protocole non fiable mais qui permet une communication rapide.

Exemple d'applications utilisant la communication UDP :

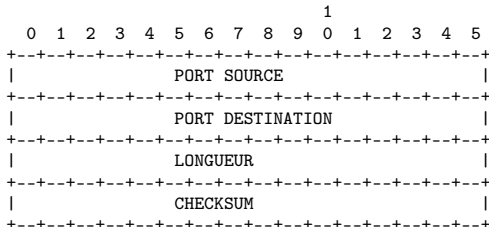
- applications temps réel lorsque la communication doit être rapide et fluide plutôt que sans erreur.
- applications multicast (multi-diffusion) : diffusion à un ensemble d'abonnés
- protocoles de routage pour les messages d'actualisation. Par exemple le protocole Routing Information Protocol (RIP)

# Communication UDP

Un paquet UDP est encapsulé dans un paquet IP :



L'entête UDP a le format suivant :



# Communication UDP

## paquet UDP

- Si  $CHECKSUM \neq 0 \Rightarrow$  on peut vérifier l'intégrité du paquet UDP (en-têtes et données)
- $LONGUEUR$  contient la taille de l'en-tête UDP + données  $\leq 65\,535$   
 $\Rightarrow$  tailles des données  $\leq 65\,527$  octets

En pratique,

- la MTU (Maximal transmission Unit) entre deux routeurs est la taille maximale d'un paquet pouvant être transmis en une fois entre ces deux routeurs  
MTU en octets :    IPv4 > 68    IPv6 > 1280    Ethernet, ADSL, Wifi  $\simeq 1500$
- si le paquet emprunte un trajet sur lequel la MTU entre deux routeurs est plus petite que la taille du paquet, celui-ci est fragmenté en plus petits paquets par le premier routeur. Ces petits paquets sont ensuite réassemblés par le deuxième routeur  $\Rightarrow$  augmente la probabilité de perte de paquets et de paquets erronés
- il est donc préférable de prévoir des tailles de données pas trop grande

# Communication UDP

## client UDP

Les étapes du client UDP :

- ❶ créer la socket UDP,
- ❷ le client peut alors
  - envoyer des paquets UDP à une adresse et un port donnés,
  - **après un 1er envoi**, la socket est automatiquement liée à un port local **P**, et peut alors réceptionner des paquets UDP sur ce port **P**.

# Communication UDP

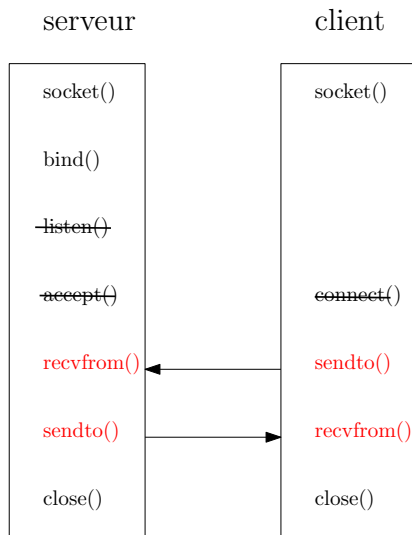
## serveur UDP

Les étapes du serveur UDP :

- ❶ créer la socket serveur UDP,
- ❷ lier la socket serveur à un port  $Q$  d'écoute,
- ❸ le serveur peut alors
  - réceptionner les paquets UDP reçus sur le port  $Q$ ,
  - envoyer des paquets UDP à une adresse et un port donnés.

# Communication UDP

client/serveur UDP





# Communication UDP

créer une socket UDP

Une socket UDP en C doit être déclarée de type `SOCK_DGRAM`.

En IPv6 :

```
int sock = socket(PF_INET6, SOCK_DGRAM, 0);
```

En IPv4 :

```
int sock = socket(PF_INET, SOCK_DGRAM, 0);
```

# Communication UDP

## envoyer des paquets UDP

Pour envoyer des paquets UDP à une adresse de destination donnée, on utilise la fonction

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

- `sockfd` : socket de communication UDP
- `buf` : tampon de suite d'octets constituant le message à envoyer
- `len` : nombre d'octets à envoyer
- `flags = 0`
- `dest_addr` : adresse de destination et sa taille `addrlen`
- retourne le nombre d'octets envoyés si succès, `-1` sinon (voir `errno`)

# Communication UDP

un client UDP IPv6

```
int main(int argc, char *argv[]){

    int sock = socket(PF_INET6, SOCK_DGRAM, 0);
    if (sock < 0) return -1;

    //adresse de destination
    struct sockaddr_in6 adr;
    memset(&adr, 0, sizeof(adr));
    adr.sin6_family = AF_INET6;
    inet_pton(AF_INET6, argv[1], &adr.sin6_addr);
    adr.sin6_port = htons(atoi(argv[2]));
    socklen_t len = sizeof(adr);

    char buffer[BUF_SIZE];
    sprintf(buffer, "Conf sur les start-up mardi 01 avril!\n");

    int env = sendto(sock, buffer, strlen(buffer), 0, (struct sockaddr *)&adr, len);
    if (env < 0){
        perror("echec de sendto.");
        return -1;
    }
}
```

# Communication UDP

## un client UDP

- Pas besoin de l'étape de connexion avec `connect` !

**Conséquence** : on ne doit pas oublier pour l'envoi

- de préparer l'adresse de destination : IP + PORT
- de passer son adresse mémoire à `sendto`

C'est le modèle de la carte postale formée d'un message et de l'adresse du destinataire : **Nom** + **Adresse** ↔ **PORT** + **IP**.

- après le `sendto`, la socket `sock` est liée à un port local et peut recevoir des paquets sur ce port.
- En IPv4, c'est pareil excepté qu'il faut déclarer une socket UDP IPv4 et utiliser des adresses de type `struct sockaddr_in`.

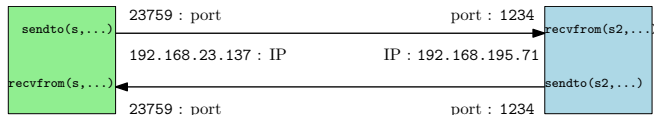
# Communication UDP

## un client UDP IPv4

```
int main(int argc, char *argv[]){
    int s = socket(PF_INET, SOCK_DGRAM, 0);
    if (s < 0) return -1;

    struct sockaddr_in adr;
    memset(&adr, 0, sizeof(adr));
    adr.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.195.71", &adr.sin_addr);
    adr.sin_port = htons(1234);
    socklen_t len = sizeof(adr);

    char buffer[BUF_SIZE];
    sprintf(buffer, "Conf sur les start-up mardi 01 avril!\n");
    int env = sendto(s, buffer, strlen(buffer), 0, (struct sockaddr *)&adr, len);
    if (env < 0) return -1;
    ...
}
```



# Communication UDP

netcat

Comment tester notre client ?

La commande **telnet** ne permet que les communications TCP.

On utilise donc la commande **netcat** avec l'option **-u** pour des communications UDP.

La commande pour simuler un serveur IPv6 qui attend des paquets UDP sur le port 9898 est donc :

```
$ nc -6ul 9898
```

# Communication UDP

## réception de paquets UDP

Pour **recevoir** des paquets UDP sur un port **P** donné, une **socket** UDP doit être **liée** à ce **port**. Il y a deux situations :

- soit la socket UDP n'est liée à aucun port, et il faut commencer par la lier au port **P** avec **bind**,
- soit la socket UDP est déjà liée au port **P**, parce qu'elle a auparavant envoyé un paquet UDP depuis le port **P** ou a déjà fait un **bind**.

Différence avec TCP :

- en TCP, après le handshake ( **connect/accept** ), le serveur ou le client peuvent initier la conversation en envoyant le premier message
- en UDP, le client doit initier la conversation en envoyant le premier message

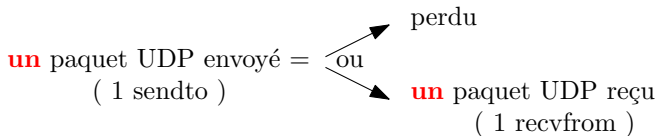
# Communication UDP

## réception de paquets UDP

Pour **recevoir** des paquets UDP sur un port **P** donné, une **socket** UDP doit être **liée** à ce **port**. Il y a deux situations :

- soit la socket UDP n'est liée à aucun port, et il faut commencer par la lier au port **P** avec **bind**,
- soit la socket UDP est déjà liée au port **P**, parce qu'elle a auparavant envoyé un paquet UDP depuis le port **P** ou a déjà fait un **bind**.

Différence avec TCP :





# Communication UDP

## réception de paquets UDP

Ensuite, pour la réception, on utilise la fonction

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

- `sockfd` : socket de communication UDP
- `buf` : tampon de taille `len` pour stocker le message
- `flags = 0`
- `src_addr` : adresse mémoire de l'adresse de la source du message remplie lors de l'appel → `NULL` si on ne veut pas stocker l'adresse.
- `addrlen` : adresse de la taille de `src_addr`. Doit être remplie avant l'appel et peut-être modifiée lors de l'appel → `NULL` si on ne veut pas stocker l'adresse.
- retourne le nombre d'octets reçus si succès, `-1` sinon (voir `errno`). Peut retourner la valeur `0` si le message est vide.

# Communication UDP

## un serveur IPv6 UDP

```
int main(int argc, char *argv[]){
    int sock = socket(PF_INET6, SOCK_DGRAM, 0);
    if (sock < 0) return -1;

    struct sockaddr_in6 servadr;
    memset(&servadr, 0, sizeof(servadr));
    servadr.sin6_family = AF_INET6;
    servadr.sin6_addr = in6addr_any;
    servadr.sin6_port = htons(atoi(argv[1]));

    if (bind(sock, (struct sockaddr *)&servadr, sizeof(servadr)) < 0) return -1;

    char buffer[BUF_SIZE+1];
    struct sockaddr_in6 cliadr;
    socklen_t len = sizeof(cliadr);

    memset(buffer, 0, BUF_SIZE+1);
    int r = recvfrom(sock, buffer, BUF_SIZE, 0, (struct sockaddr *)&cliadr, &len);
    if (r < 0) return -1;
    printf("message reçu - %d octets : %s\n", r, buffer);

    close(sock);
    return 0;
}
```

# Communication UDP

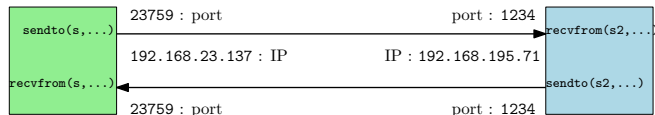
## un serveur IPv4 UDP

```
int main(int argc, char *argv[]){
    int s2 = socket(PF_INET, SOCK_DGRAM, 0);
    if (s2 < 0) return -1;

    struct sockaddr_in servadr;
    memset(&servadr, 0, sizeof(servadr));
    servadr.sin_family = AF_INET;
    servadr.sin_addr = htonl(INADDR_ANY);
    servadr.sin_port = htons(1234);

    if (bind(s2, (struct sockaddr *)&servadr, sizeof(servadr)) < 0) return -1;

    char buffer[BUF_SIZE+1];
    struct sockaddr_in cliadr;
    socklen_t len = sizeof(cliadr);
    memset(buffer, 0, BUF_SIZE+1);
    if (recvfrom(s2, buffer, BUF_SIZE, 0, (struct sockaddr *)&cliadr, &len) < 0) return -1;
    ...
}
```



# Communication UDP

## un serveur UDP

Le serveur peut recevoir des messages de différents clients sur son port 1234.

Le serveur récupère l'*adresse du client* et sa longueur (5<sup>ème</sup> et 6<sup>ème</sup> arguments de `recvfrom`) à *chaque réception* de message :

- s'il veut *identifier* l'expéditeur grâce à son IP et éventuellement port,
- ou s'il veut *renvoyer un message* à l'expéditeur.

# Communication UDP

## un serveur UDP

Le serveur peut recevoir des messages de différents clients sur son port 1234.

Le serveur récupère l'*adresse du client* et sa longueur (5<sup>ème</sup> et 6<sup>ème</sup> arguments de `recvfrom`) à *chaque réception* de message :

- s'il veut *identifier* l'expéditeur grâce à son IP et éventuellement port,
- ou s'il veut *renvoyer un message* à l'expéditeur.

1 socket TCP liée à un port donné : 1 pair  $\leftrightarrow$  1 pair

1 socket UDP liée à un port donné : 1 pair  $\leftrightarrow$  plusieurs pairs

# Communication UDP

un serveur IPv6 UDP polyvalent

Afin que le serveur accepte les paquets UDP sur IPv6 et IPv4, il suffit comme pour TCP, de demander à ce que la socket IPv6 ne soit pas en mode `IPV6_ONLY`. Il faut toujours le faire avant de lier la socket (`bind`).

```
int optval = 0;
int r = setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &optval, sizeof(optval));
if (r < 0)
    perror("erreur : IPv4 impossible");

if (bind(sock, (struct sockaddr *)&servadr, sizeof(servadr)) < 0) return -1;
```

Les adresses IPv4 sont alors transformées en interne au serveur, en adresses IPv4-mapped.

# Communication UDP

## réception de paquets UDP

- Comme on a récupéré l'adresse du client dans `cliadr`, on peut maintenant lui envoyer des messages avec `sendto`

```
sendto(sock, buffer, strlen(buffer), 0, (struct sockaddr *)&cliadr, len);
```

- Si on ne souhaite pas récupérer l'adresse du client, on peut faire l'appel

```
int r = recvfrom(sock, buffer, BUF_SIZE, 0, NULL, NULL);
```

qui est équivalent à l'appel

```
int r = recv(sock, buffer, BUF_SIZE, 0);
```

# Communication UDP

## réception de paquets UDP

Mais comment sait-on quand arrêter d'attendre la réception de messages ?

- Le protocole doit spécifier un format qui permet de détecter la fin de la réception.

*Par exemple, un dernier paquet vide (sans données) ou terminant par une chaîne de caractères spécifique.*

- Mais cela ne suffit pas toujours.
  - On a déjà dit que lors de la réception, on ne peut pas être sûr de réceptionner tous les paquets envoyés.
  - Par ailleurs, la fonction `recvfrom` est bloquante par défaut.

Cela signifie que si une application pense recevoir 10 messages et que certains n'arrivent jamais, elle va rester bloquée indéfiniment sur le `recvfrom`.



# Communication UDP

## réception de paquets UDP

Que peut-on faire ?

- déléguer à un processus léger la réception des messages UDP sur un port donné. C'est moyennement satisfaisant car si l'attente est infinie, il vaudrait mieux la terminer,
- déléguer à un processus (`fork`) la réception des messages UDP sur un port donné et tuer le processus si celui-ci ne termine pas après un certain temps,
- surveiller les descripteurs d'entrées-sorties (sera vu dans un cours ultérieur).

# Communication UDP

## réception de paquets UDP

Le protocole pourrait prévoir que l'expéditeur envoie des paquets de fin jusqu'à recevoir un acquittement, mais le problème se reporte alors sur l'expéditeur...

Attention, on est **pas en mode connecté**. Si `recv` ou `recvfrom` retourne `0`, cela signifie en *UDP*, qu'elles ont reçu un **paquet vide** mais pas forcément que l'hôte distant a terminé (sauf si le protocole de l'application le spécifie).

# Communication UDP

adresse de noms

Si l'adresse IP d'expédition est donnée sous forme d'adresse de noms, on fait presque comme en TCP, on utilise `getaddrinfo` avec :

- le champs `ai_socktype` de `hints` doit être égal à `SOCK_DGRAM`
- il n'y a pas de connect dans la boucle de parcourt des adresses disponibles
- si on veut un client polyvalent (IPv4/IPv6) :
  - on peut utiliser une socket polymorphe en mettant le champs `ai_flags` de `hints` à `AI_V4MAPPED|AI_ALL`,
  - ou mettre le champs `ai_family` de `hints` à `AF_UNSPEC`.

# Communication UDP

adresse de noms

```
struct addrinfo hints, *r, *p;
int sock, res;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_DGRAM;

if ((res = getaddrinfo(hostname, port, &hints, &r)) != 0 || NULL == r)
    fprintf(stderr, "echec getaddrinfo : %s\n", gai_strerror(res));
else{
    p = r;
    while( p != NULL ){
        if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0)
            break;
        close(*sock);
        p = p->ai_next;
    }
}
struct sockaddr_in6 addr;
if(p != NULL)
    addr = *((struct sockaddr_in6 *) p->ai_addr);
freeaddrinfo(r);
```

# Communication UDP

client polyvalent

```
struct addrinfo hints, *r, *p;
int sock, res;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;

if ((res = getaddrinfo(hostname, port, &hints, &r)) != 0 || NULL == r){
    fprintf(stderr, "echec getaddrinfo : %s\n", gai_strerror(res));
    return -1;
}

p = r;
while( p != NULL ){
    if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0)
        break;
    close(sock);
    p = p->ai_next;
}

if (p == NULL) return -2;

//on stocke l'adresse de connexion et sa longueur
struct sockaddr_in6 addr = *((struct sockaddr_in6 *) p->ai_addr);
socklen_t adrlen = p->ai_addrlen;

//on libère la mémoire allouée par getaddrinfo
freeaddrinfo(r);
```

# Communication UDP

client polyvalent

Attention, on stocke l'adresse dans une `struct sockaddr_in6` car une adresse IPv6 prend plus de place qu'une adresse IPv4. La longueur de l'adresse `adrrlen` rend compte de la longueur réelle de l'adresse.

IPv4

p->ai\_addr



0xa4ef6b12

0xa53c1cd5



↑  
&addr

IPv6

p->ai\_addr



0xa4ef6b12

0xa53c1cd5



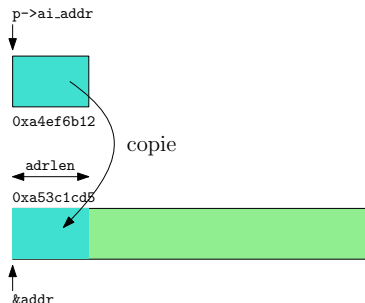
↑  
&addr

# Communication UDP

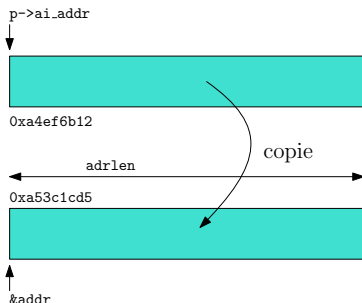
client polyvalent

Attention, on stocke l'adresse dans une `struct sockaddr_in6` car une adresse IPv6 prend plus de place qu'une adresse IPv4. La longueur de l'adresse `adrrlen` rend compte de la longueur réelle de l'adresse.

IPv4



IPv6



# Communication UDP

client polyvalent

Lorsqu'on libère la liste chaînée des adresses `r`, l'adresse de l'hôte distant et sa longueur sont sauvegardées ailleurs en mémoire.

